RC 21430 (96745) March 15, 1999
Computer Science/Mathematics

IBM Research Report

Algorithm-based Software Design for Multi-Project Use

M. Trumbo[1] J. Kalagnanam[2] H. S. Lee[3].

IBM Research Division
Almaden · Austin · T.J. Watson · Beijing · Haifa · Tokyo · Zurich

[1]IBM Canada, 2220 Walkley Road, Ottawa, Ontario, K1G 5L2, trumbo}@us.ibm.com

[2]IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York 10598, jayant}@us.ibm.com

[3]IBM Global Services, Yorktown Hts, NY, leehs}us.ibm.com

Abstract

We present a design methodology for developing algorithm based software platforms for multi-project use with a focus on production planning and scheduling applications. The methodology leads to a partitioning and decoupling of software design activities by using abstraction and complexity hiding in a layered architecture. The abstractions are centered around high level generalizations of common problem classes in Operations Research such as *packing, sequencing* and *grouping*. Complexity hiding is achieved by using minimal informational interfaces between various levels in the architecture. The resultant design separates the customer specific information from the abstract problem representations and algorithms and thereby facilitates reuse. In addition, such a separation delineates the intellectual property of the customer from that of the solution provider, an important prerequisite for reuse in a business setting.

Keywords: Software design, production planning and scheduling, abstraction, complexity hiding, software reuse, intellectual property.

1 Introduction

In this paper, we address common problems encountered when designing software solutions for operations management problems. Such solutions have at their heart algorithms to solve a class of operations management problems usually requiring optimization. One must determine

- how to design the software so as to maximize the fraction of the software that can be reused to solve other instances of the same problem class,
- how to protect the intellectual property which resides in the algorithmic solutions and facilitate reuse for solving other instances of the operations management problem with other clients, and
- how to partition software development to maximize resource utilization and schedule.

These issues are of critical importance in today's business environment where short project lead times and slim profit margins call for efficient use of resources. Software implementation often accounts for the largest part of project cost and hence software reuse across multiple projects is of critical importance. In addition, separation of the intellectual property of the solution provider and the client in the software implementation is critical for providing cost-effective solutions for multiple clients. Software implementations where the intellectual property of the solution provider is not easily separable from that of the client might make it difficult to reuse the same code for a new client due to confidentiality and copyright agreements. Finally, for multi-project use, the fact that each client's business process is so different means that the focus of the reuse effort should be on the algorithms that solve the client problems, not the implementation of the client business objects.

We address these general problems in the context of several recent efforts at IBM T.J. Watson Research Center for developing production planning and scheduling solutions for manufacturing environments such as steel and paper industry (Kalagnanam et al 98, Murthy et al 98). Applications in both these domains have demonstrated a need for building a software library of problem classes and algorithms with specific goals of rapid prototyping and reuse. The design principles that we have developed and used to build such a software platform are presented in this paper. The fundamental strategy used to build such a platform involves

- identifying high level abstractions of common problem classes encountered in production planning and scheduling, and
- developing layers of transformations to move from client specific problems to these high level abstractions to implementations of algorithms for instances of the problem class.

The rest of the paper is organized as follows. Section 2 provides a discussion of the motivations and the design issues in building a reusable software platform for operations management problems. Section 3 provides the design methodology that we have developed and used to build such a software platform. Section 4 provides an overview of such a platform built around the class of packing problems. Section 5 presents an industrial strength example (the Surplus Inventory Matching Problem) that illustrates the design principles outlined in sections 2, 3. Section 6 concludes with a description of three software development roles in developing application software and a set of recommendations for each role.

2 Design Goals

The main design issue with developing such a software platform relates to its reuse in solving a new instance of a problem class. The primary design goals are as follows:

- A software platform should allow for rapid prototyping with maximal reuse of available problem representations and algorithms. This implies that
  - constraints and objectives evaluations are implemented once per particular problem instance,
  - upper bounds and lower bounds can be derived using available algorithms on relaxations of the problem instance, and
  - we should be able to generate quick approximate answers to problem new problem instances
- Abstract problem representations from the problem solver thereby separating the problem specifics from the algorithms while still allowing the algorithms to solve the problem nearly optimally.
- Seamlessly adopt new algorithms form other sources (such as those that can be downloaded from the internet) into the platform with minimal customization?

2.1 Design Principles

In this section we outline a set of general software design principles which allow us to achieve the design goals that we have outlined for building reusable software platforms. These general design principles are:

- Provide an infrastructure for embedding intellectual assets by *abstraction and generalization* thus moving the structured design process up to a cross project level.
- Optimally partition and decouple software design activities in order to maximize efficiency by *abstraction and complexity hiding*
- Promote maximum reusability by *implementing the minimal interface required* and by *designing for uncertainty*

Minimal interface provides flexibility to use generalization and makes for the most adaptable system. The notion of building a minimal interface is motivated by a question regarding information content required by a piece of reusable software. For example, does an algorithm really need to know why a particular constraint is satisfied or violated or just whether if it is satisfied or violated.

Abstraction and complexity hiding allows the separation of the design and implementation activities. Such a separation is also useful to separate the intellectual property of the solution provider and that of the customer/client. Usually, the intellectual property of the solution provider resides in the abstract representations and algorithms while that of the customer resides in the complex constraints and specifications.

Designing for uncertainty means assuming that the specifications are going to change and that correctness is the number one goal. Hence, arbitrary changes to any of the constraints and specifications will be remain isolated from the algorithm implementations and the abstract solution class will remain correct.

Generalization is the basis for building hierarchical representations of knowledge and the structured embedding of assets. Moreover hierarchical knowledge representation facilitates rapid prototyping by inheriting at the right level in the hierarchy.

2.2 Definition of success

To evaluate the success of building a software platform we need a set of measures. In this subsection we outlines some useful measures to evaluate software designs. In general, we want to minimize the number of adapter classes and algorithm implementations that we write. We can do this by minimizing the number of higher level representations.

Say we have N real world problems, n more abstract, generic problems, and M algorithms. We want to avoid having to do NM algorithm implementations. Basically want $nM + nN$ to be much smaller than NM which means we have to minimize the number of abstract problems, n. We do this by choosing these abstractions well. Probably most savings come from implementing algorithms. That savings is $NM - nM$ algorithm implementations.

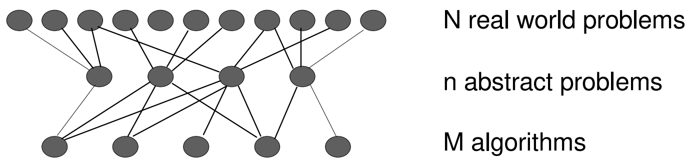

Figure 1: Minimizing overall software development effort via abstractions

Of course, the network above is not fully connected, and so we can't really use this as a precise measure so we will have to also rely on anecdotal evidence gathered regarding the ease of use and reduced development time.

Finally, some measures of OO code reuse can also be applied to determine how well we are doing.

3 Overview of the Design Methodology

We now present a general design methodology for developing algorithm based software platforms for multi-project use. This methodology has been developed in the context of building software platforms over several application projects in the production planning and scheduling domain. The fundamental thrust of this methodology is to a partition and decouple the software design activities so as to separate the customer specific implementations from the more general problem representations and algorithms which can then be reused across projects. The fundamental strategy used to build such a platform involves

- identifying high level abstractions of common problem classes encountered in production planning and scheduling, and
- developing layers of transformations to move from client specific problems to these high level abstractions to implementations of algorithms for instances of the problem class.

In this section we provide an overview of some abstractions used to model problems in the OR literature. This overview is followed by a discussion of how to design a layered architecture to adapt real world problems to these abstractions.

3.1 Abstract Operation Research Problems

While developing production planning and scheduling solutions for manufacturing domains, time and again we have encountered problems that are very similar and differ only in their constraints. A fundamental observation is that problem classes can be organized around problem abstractions and different problems within a problem class differ only in their constraints. To build a software platform, it is important to clearly identify the broad categories of problems that we want in our library. Therefore, we have grouped the most commonly encountered production planning and scheduling problems in to the following three categories. Further details of reasoning behind the classification can be found in (Kalagnanam, Murthy, and Miller 1997).

- **Sequencing** Scheduling problems typically require sequencing jobs on one or more machines. The traveling salesman problem (TSP), vehicle routing problem (VRP), job shop scheduling problem, flow shop scheduling problems are variations of the sequencing problem. Typical applications are encountered in job shop scheduling.
- **Packing** Inventory matching problems and job allocation (to machines) problems both involve packing problems. The inventory matching problem involves applying an order book against an existing inventory and can be treated as a constrained multiple knapsack problem where the orders are items to be packed into multiple knapsacks constrained by various manufacturing considerations. Similarly, the allocation of jobs to machines in a job scheduling problem is analogous to the multiple knapsack problem where jobs are analogous to the items and machines are made analogous to the knapsacks and the packing is once again constrained by compatibility constraints.
- **Grouping** Examples include grouping of jobs of same kind together in to batches in job scheduling problem, and finding the best combination of items in the groups for instances of cutting stock problems such as trimming. These class of problems are typically constrained bin packing problems with a wide variety of manufacturing constraints.

These categories emerge because the underlying objects in each category have similar behaviors. In the case of sequencing, for example, the underlying structure is a sequence of objects between which are defined hard and soft constraints which indicate the set of items that can be sequenced after one another. In the case of matching, the input objects consist of items and knapsacks between which are defined a set of assignment constraints.

3.2 A High-level Layered Software Design Template

The utility of high-level abstractions is obvious in the OR literature. When an author writes about a traveling salesperson problem (TSP), the reader immediately has a fairly accurate idea of what problem is being discussed. On the other hand, it is easy to find several different implementations of for each existing algorithm for solving the TSP, each with its own interface, and . For an organization whose goal to make money by solving real world problems using algorithms found in the literature, this is less than optimal. What is lacking is a standard, flexible, and efficient interface for the TSP.

Our design approach is *algorithm-based* in that we build libraries of algorithms and their corresponding abstractions simultaneously, keeping in mind that there are classes of problems that are related.

We illustrate the above point with a Traveling Salesman Problem (TSP) example, a common problem in sequencing. When we design the `TSPProblem` object, we separate the algorithm implementation from the problem representation. For example, the `TSPSolver` is separate from the problem representation `TSPProblem` which it requires as input to solve an instance of the TSP. The primary question here is whether we have gained any advantages in terms of reuse by doing this. The advantages become apparent when we consider the Prize Collecting TSP (PCTSP) a variant of the TSP with an additional constraint which specifies that the minimum prize level required of any feasible solution. We once again separate the representation in the `PCTSPProblem` object from its solver `PCTSPSolver`. However, notice that the `PCTSPProblem` is derived from the `TSPProblem` with an additional interface for the prize associated with each city. This hierarchical representation allows the `PCTSPProblem` to be solved as a `TSPProblem` without any additional code by reusing the `TSPSolver`. Clearly this solution would have ignored the prize constraints and hence would not be feasible. However, often such relaxed solutions form a starting point for designing heuristics and hence can be useful in practice. We **do not** try to simultaneously try to model the PCTSP problem. Instead we derive the PCTSP problem from the TSP as follows.

```
class TSPProblem
{
public:
  virtual int numberOfCities() const = 0;
  virtual double distance(int city1, int city2) const = 0;
  virtual void makeSubTour(int city1, city2) = 0;
};

class PCTSPProblem : public TSPProblem
{
public:
  virtual int prize(int city) const() = 0;
};

void TSPSolver(TSPProblem & tspProblem);
void PCTSPSolver(PCTSPProblem & pctspProblem);

Class MyTSPProblem : public TSPProblem { ... };
Class MyPCTSPProblem : public PCTSPProblem { ... };
```

The utility of this approach becomes evident when we have multiple similar problems.

```
int
main(int argc, char * argv[])
{
  MyTSPProblem myTSPProblem;
  MyPCTSPProblem myPCTSPProblem;
  // Solve the TSP problem instance using a TSP solver.
  TSPSolver(myTSPProblem);
  // Solve the PCTSP problem instance using a TSP solver. This
  // works, but since it ignores the prize it will generate a bad solution.
  TSPSolver(myPCTSPProblem);
  // Solve the PCTSP problem instance using a PCTSP solver.
  PCTSPSolver1(myPCTSPProblem);
  // Try to solve the TSP problem with a PCTSP solver. This won´t even
  // compile! The PCTSP needs to know the prize on a city.
  PCTSPSolver1(myTSPProblem);
  return 0;
}
```

The software that implements algorithms for the TSP problem is unaffected by the introduction of the PCTSP problem. Additionally since the PCTSP is a TSP it can still be solved by the TSP algorithms which are unaware of the notion of a prize associated with a city, ie, a relaxation of the PCTSP. This provides a natural way to organize useful problem hierarchies which model the relaxations between problems within a problem class.

3.3 The Methodology

Figure 2 shows a general approach to organizing algorithms and abstractions and linking them to real world problem and to each other via adapters.

At the highest level of abstraction each algorithm has an associated abstract problem. The abstract problem specifies the minimum interface required by the algorithm to solve the problem. Some problems are generalizations of other problems and are related by inheritance. We may also be able to translate between abstract problems using an adapter. At the lowest level we have the individual client problem classes.

The middle layers connect, via adapters, the client problems to the high-level abstractions so that the client problems can be solved using the existing algorithms. Hence the algorithms are independent of the client-specific problem classes, being separated by adapters. This allows algorithms and their abstraction, middle-level abstractions, and client-specific code to proceed simultaneously and relatively independently.

3.4 Bootstrapping the Platform

The design approach described in the previous section makes sense if an organization is designing from scratch. This is usually not the case. An organization may have a collection of algorithms

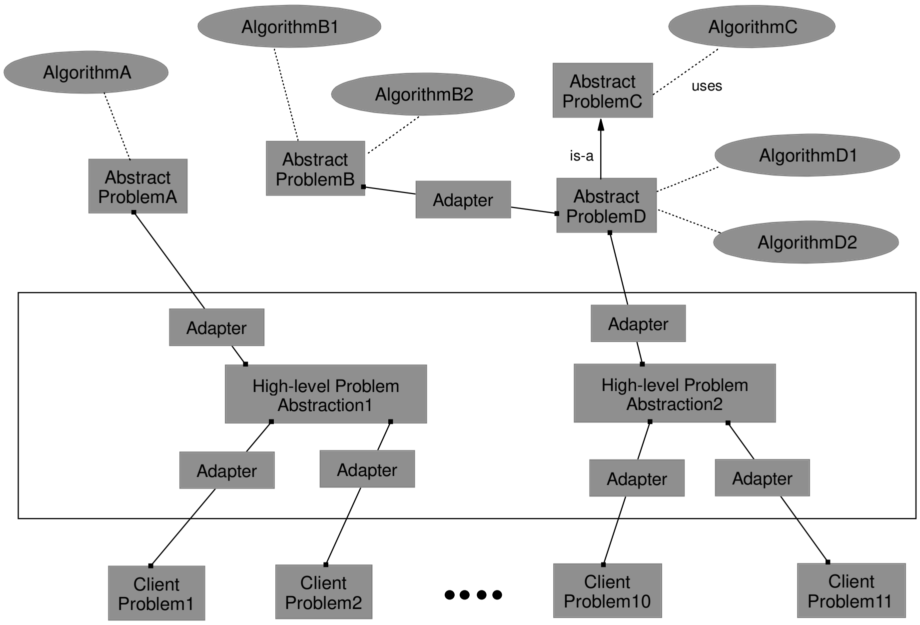

Figure 2: Linking customer-specific problems to high-level abstractions

and a collection of representations. It is likely that the algorithms operate on representations whose interfaces are too broad, ie, any particular algorithm operates on a object that provides, via its interface, more information than the algorithm actually needs. This hinders reuse. The first step is to define minimal interfaces for the algorithms, freeing them from the fat interfaces to which they are bound

In Figure 3, the algorithm turned out to be an implementation of a standard OR algorithm but it does not necessarily have to be the case. It may be that the algorithm solves a new but general class of problems.

```
SpecificProblemWithNewProblemClassFlavor
  <--uses--
    SpecificAlgorithmThatSolvesNewProblemClassProblem
```

to

```
SpecificProblemWithNewProblemClassFlavor
  <--contains--
    NewProblemClassToSpecificProblemWithNewProblemClassFlavorAdapter
      --isa-->
```

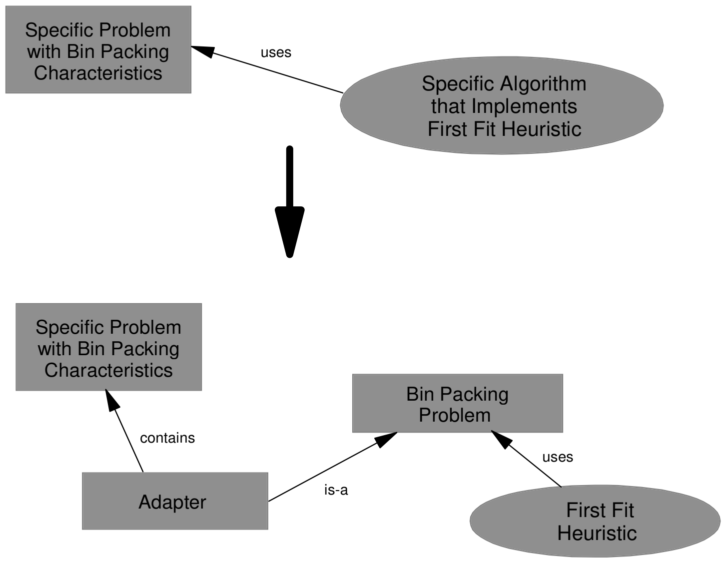

Figure 3: Bootstrapping the Platform

```
NewProblemClassProblem
  <--uses-- NewProblemClassAlgorithm
```

At this point the organization is in a better position to reuse software.

3.5 Augmenting the Platform

The natural question that arises when encountered with a new problem instance is whether it can be modeled within the existing problem representation hierarchy or whether new representations need to be introduced. In this section we outline the basic steps for answering this question.

- Identify the minimal information required to solve the new problem, ie, the minimum set of queries required to fully characterize the problem state and the minimum set of commands to change the solution state.
- Search the set of existing problems to identify if there exists a problem class that maximally satisfies the information requirements of the new problem. In some cases an existing problem class completely satisfies the requirements and in other cases, only partially. There may be several that satisfy the requirements to the same degree. Write one or more adapters between the existing problem class(es) and the new problem.

- Solve the problem using the available algorithms. If the solution is not acceptable and if the new problem was mapped into an existing problem with information loss, then write a specific algorithm to solve the new problem class.

4 The Packing Abstraction

In this section we discuss in detail the abstractions that we have developed for the packing problem class. The problem hierarchy and the related algorithms and adapters have been developed over a set of applications in steel and paper industry. The problem representation hierarchy represents a consolidation of several applications and is clearly evolving as more applications are encountered.

4.1 Problem Hierarchy

Packing problems can be modeled as multiple knapsack problems (Martello and Toth 84). A multiple knapsack problem consists of N items of weight W_i which are to be packed into M knapsacks of capacity C_i. The objective is to pack items into knapsacks so as to maximize total assigned weight. Note that the total weight packed into any knapsack is constrained by its capacity.

A number of variations to the multiple knapsack problem arise in real world applications and they differ in their packing constraints. We will outline three classes of constraints encountered in our applications:

4.1.1 Assignment Restrictions

Not all items can be assigned to all knapsacks and such restrictions lead to a *sparse bigraph*, i.e. not all edges are present in the above graph. Such restrictions allow a subset of the total items to be assigned to any knapsack. We call this class Multiple Knapsack with Assignment Restrictions (MKAR).

4.1.2 Color Constraints

Additional constraints are specified in terms of color constraints which restrict the total number of distinctly colored items assigned to a knapsack to two. Notice that we have indicated the color of an item using a fill pattern. The third knapsack from top has three items of different colors not all of which can be packed together. We call this class Multiple Knapsack with Color Constraints (MKC).

4.1.3 Geometric Constraints

While packing multiple items in a knapsack, the 2-D layout of the items poses geometric constraints on packing and leads to a cutting stock problem for each knapsack. We call this class Multiple Knapsack with 2-D constraints (MK2D).

We have built a problem class hierarchy around these three problem classes as a initial point in building a hierarchy of problem representations for packing problems.

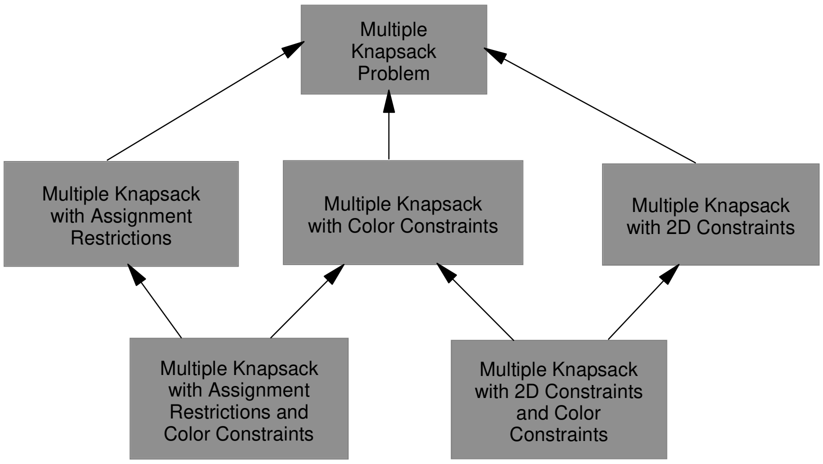

Figure 4: Problem Hierarchies for Packing Problems

Next we describe the class prototypes for this problem hierarchy and explicate the information requirements of each class.

```
// The basic multiple knapsack problem

class MultipleKnapsack
{
public:
  virtual Iterator<int> itemIterator() const = 0;
  virtual Iterator<int> knapsackIterator const = 0;
  virtual double knapsackWeight(int knapsackId) const = 0;
  virtual double itemWeight(int itemId) const = 0;
  virtual void doAssign(int itemId, int knapsackId) = 0;
};

// The multiple knapsack problem with assignment restrictions.

class MultipleKnapsackAR : public MultipleKnapsack
{
public:
  virtual int canAssign(int itemId, int knapsackId) const = 0;
```

```
};

// The multiple knapsack problem with color constraints is
// an multiple knapsack problem with assignment restrictions.

class MultipleKnapsackColor: public MultipleKnapsack
{
public:
  virtual int maximumNumberOfColorsPerKnapsack() const = 0;
  virtual int itemColor(int itemId) const = 0;
};

// The multiple knapsack problem with two dimensional constraints
// Question: How would an algorithm use the geometry information?

class MultipleKnapsack2D : public MultipleKnapsackAR
{
public:
  virtual double itemWidth(int itemId) const = 0;
  virtual double itemLength(int itemId) const = 0;
};

// The multiple knapsack problem with two dimensional constraints
// and color constraints.

class MultipleKnapsackColorand2D : public MultipleKnapsackColor,
                                   public MultipleKnapsack2D
{
};

class ClientProblemWithPackingFlavor
{
};

// This class adapts the interface of the ClientProblemWithPackingFlavor
// to the MultipleKnapsackColor problem interface. It makes it possible
// to solve the ClientProblemWithPackingFlavor using any algorithm
// that can solve the MultipleKnapsackColor problem, including, of
// course, those algorithms that solve the simple MultipleKnapsack
// problem.

class ClientProblemWithPackingFlavorToMultipleKnapSackProblemAdapter
   : public MultipleKnapsackColor
{
public:
  virtual Iterator<int> itemIterator() const;
  virtual Iterator<int> knapsackIterator const;
  virtual double knapsackWeight(int knapsackId) const;
  virtual double itemWeight(int itemId) const;
  virtual void doAssign(int itemId, int knapsackId);
  virtual int maximumNumberOfColorsPerKnapsack() const;
  virtual int itemColor(int itemId) const;
private:
  ClientProblemWithPackingFlavor * problem;
};
```

These class prototypes indicate the new information requirements for each new problem class in the problem hierarchy. For example assignment restrictions require the introduction mechanism for determining the knapsacks that can be used for a given item. Without assignment restrictions we assume that any item can be potentially packed into any knapsack. Similar color constraints require the introduction of the color attribute on the item and a member function that check color constraints. The modular introduction of new information requirements for each problem becomes apparent from the class headers.

4.2 Algorithms for Packing

We will introduce two algorithm classes that could be potentially used to solve the packing problems. Note that this list is not intended to exhaustive or even a complete description of algorithm library. However they are presented here as a vehicle to illustrate how the adapters are written to adapt the problem representations to the algorithms. The interested reader is referred to (Kalagnanam et al 1998) for more details on algorithms.

4.2.1 Integer Programming Approaches

The multiple knapsack consists of a set of items $N = \{1, \ldots, n\}$ and a set of knapsacks $M = \{1, \ldots, m\}$. Each item $j \in N$ has a weight w_j and each knapsack $i \in M$ has a capacity c_i associated with it. All w_j and c_i are positive real numbers. In addition, for each item $j \in N$ a set $A_j \subseteq M$ of knapsacks that can hold item j is specified. Although A_j's suffice to represent the assignment restrictions, for convenience we also specify for each knapsack $i \in M$, the set $B_i \subseteq N$ of items that can be assigned to the knapsack.

The multiple knapsack problem can be formulated as an integer problem as follows:

$$
\begin{array}{ll}
\max & \sum_{i \in M} \sum_{j \in B_i} w_j \, x_{ij} \\
\text{st} & \\
 & \sum_{j \in B_i} w_j \, x_{ij} \le c_i, \;\; i \in M \\
 & \sum_{i \in A_j} x_{ij} \le 1, \;\; j \in N \\
 & x_{ij} \in \{0, 1\}, \; i \in A_j, \; j \in N
\end{array}
$$

where the 0-1 variable x_{ij} denotes whether item j is assigned to knapsack i. The LP relaxation corresponds to relaxing integrality of these variables.

The information required to solve the multiple knapsack problem using IP is as follows

1. Integers (N, M) indicating the number of items and knapsacks,

2. Arrays of real numbers c_i and w_j which provide the capacities of the knapsacks and the weights of the items,
3. set of integers A_j which indicate the subset of knapsacks that can hold item j, and
4. a set of integers B_i which indicate the subset of items that can be assigned to knapsack i.
5. An array of strings which indicate the color of each item.
6. Assuming that geometric problem associated with packing items into the knapsack problem is 2D, we need two array of real numbers, width u_j and length l_j for each item.

Note that for the Multiple Knapsack problem, all items can be placed in any knapsack and hence the assignment restrictions denoted by A_j and B_i are *not* required. However these sets are required for the Multiple Knapsack with Assignment Restrictions.

Modeling the color constraints is more complicated and requires the introduction of new integer variables $y_j^{k(i)}$ which takes a value 1 if knapsack j obtains an item i of color k. Without getting into a detailed discussion of the formulations, we can see that an of strings which provide the color of each item is required.

Similarly modeling geometric constraints is complicated and requires a column generation approach and is outside the scope of this paper. However it is sufficient to note that the width and length information for each item is required to check the geometric constraints for packing.

The abstract problem representation for the IP depends strongly on the solver. It is not feasible to try to generate a general IntegerProgrammingProblem class, for example. What we can do is write an IP solver that operates directly on the multiple knapsack problem.

```
void
solveUsingOSL(MultipleKnapsackColor & problem)
{
   // generate matrix
   // generate constraints
   // invoke solver
}
```

When we do come across a problem like the ClientProblemWithPackingFlavor, we can use this algorithm.

4.2.2 Network based Heuristics

It is often convenient to think of the packing problem class in terms of a bipartite graph in which the left hand nodes are the items and the right hand nodes are the knapsacks (refer to

Figure 3). Note that the existence of an edge tells us that, an unapplied item can be placed in a knapsack. Once an edge is selected in the bipartite graph, some of the edges incident to the nodes at its endpoints may become infeasible and hence need to be removed.

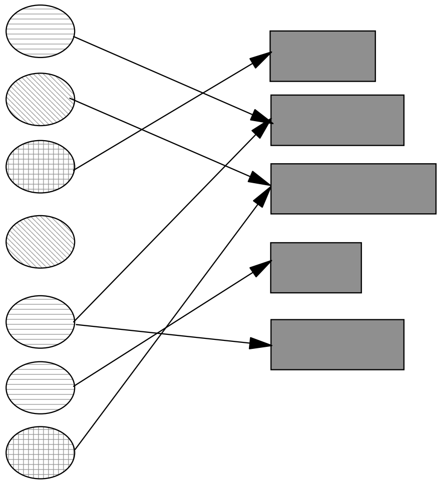

Figure 5: A bipartite graph representation of the multiple knapsack problem with color and assignment constraints

The knapsack problem can be solved heuristically by repeated application of the bipartite matching algorithm. This provides a feasible (though not optimal) solution to the problem and has been found to be quite effective in some problem instances. The bipartite graph is a graph structure which is often used as an input to the bipartite matching algorithm. The algorithm additionally requires the weight of the item nodes and the capacity of the knapsack nodes. The edges of the graph provide the assignment restrictions and the item have associated with them color attributed (shown using different fill patterns in the figure).

The edges of this graph indicate the orders which can be applied against an item in the inventory which, for concreteness, we will call a slab. The goal of inventory matching is to maximally assign the orders against slabs while minimizing waste.

The information requirements for network flow algorithms are centered around a graph representation for the multiple knapsack problem. The input requirements are specified in terms of a bipartite graph, G(V_1, V_2,E) with nodes V_1, V_2 and edges E. The edges E are specified between nodes in V_1 and V_2. Associated with each edge $e_{ij} \in E$ is an edge weight W_i which indicates the profit associated with assigning node v_i to node v_j. These requirements specify a multiple knapsack with assignment restrictions. Also associated with $v_j \in V_1$ is a color attribute c_j and the geometry in terms of the width u_j and length l_j.

The abstract problem representation for the network flow algorithms is a graph. LEDA (Melhorn

97) provides exactly such a graph abstraction and the algorithms that operate over it. What remains in order to solve the multiple knapsack problem is to adapt it to the graph problem. For example

```
class BipartiteMatchingProblem
{
public:
  virtual Iterator<int> itemIterator() const = 0;
  virtual int canLink(int item1, int item2) const = 0;
  virtual int doLink(int item1, int item2) = 0;
};

void
LEDABipartiteMatchingSolver(BipartiteMatchingProblem & problem)
{
  // Induce a graph by calling canLink on all pairs
  // Run bipartite matching on induced graph
  // Commit the edges using doLink
}

class MultipleKnapsackColorToBipartiteMatchingProblemAdapter
  : public BipartiteMatchingProblem
{
public:
  // Iterates over all items. Note that in this case, items and
  // and knapsacks cannot have the same identifiers.
  Iterator<int> itemIterator() const;
  // Queries the internal state of the problem and determines
  // if the matching is feasible given the color constraint.
  virtual int canLink(int item1, int item2) const;
  // Assign the item to the knapsack.
  virtual int doLink(int item1, int item2);
private:
  MultipleKnapsackColor * problem;
};
```

5 Case Study: The Surplus Inventory Matching Problem in the Steel Industry

We now discuss a case study where we applied the design methodology that we have outlined in the earlier sections to a real world industrial strength application in the steel industry. We describe the surplus inventory matching problem and the main elements of the software design used to implement this solution. The problem class used to model the problem and the separation of the client specific code from the general aspects of the problem model are presented. This software has been deployed in a large Japanese steel plant and is in daily use.

5.1 Background

Steel manufacturers produce slabs, coils and sheets of steel usually on a make-to-order basis. However, surplus items can be produced for three reasons.

- Surplus materials are produced to satisfy restrictions on machines and operations,
- materials are found to be out of specification during production, and
- orders are canceled.

Whereas some amount of inventory is desirable to accommodate rush or regularly placed orders, in many circumstances, inventory not only has a negative impact on cash flow, it also requires storage space and carrying costs. Hence, from the viewpoint of mill operation, it is important to reduce inventory by using as much of the inventory as quickly and judiciously as possible.

The problem of applying orders against an existing surplus inventory has a strong flavor of a packing problem – we call it the *Surplus Inventory Matching Problem*, SIMP. The number and type of orders that can be applied against a slab are constrained based on various manufacturing considerations. The goal of SIMP is to maximize the total weight of the surplus inventory that is applied against the order book. An additional goal is to minimize the unused weight of those slabs that are matched against orders. This problem can be formulated as a sparse multiple knapsack problem with color constraints. A detailed discussion of this problem is provided in (Kalagnanam et al 1998).

5.2 Problem Representation

It is most convenient to think of SIMP in terms of a bipartite graph in which the left hand nodes are the orders and the right hand nodes are the inventory items (refer to Figure 5). The edges of this graph indicate the orders which can be applied against the inventory which, for concreteness, we will call slabs. Notice that not all orders can be applied against a slab and such restrictions lead to a sparse bigraph. Additional constraints are specified in terms of color constraints which restrict the total number of distinctly colored orders[4] applied against any slab to two. This problem can be modeled as the multiple knapsack problem with assignment restrictions and color constraints, MKCAR.

We have already discussed the design and behavior of adapters used to interface the MKCAR with network flow algorithms and integer programming. The details regarding algorithms is discussed in (Kalagnanam et al 1998). The main aspects of the software design that are special interest in studying a specific application is the interface between MKCAR and SIMP.

[4]Each distinct color represents a distinct route through the processing plant.

5.3 Software Design for SIMP

The SIMP class described in the appendix is a high level problem class. Following the design principles, the SIMP interface specifies the minimum information required to query the state of a surplus inventory matching problem and to change the state into the solved state.

To solve the abstract SIMP problem, we write an adapter from SIMP to MKCAR. The adapter derives from MKCAR and contains a reference to a SIMP.

To deploy SIMP for a particular client we write an adapter class that derives from the SIMP class and contains instances of the client-specific classes required to implement the SIMP interface. Note that we could have adapted the ClientProblem directly to the MKCAR but using inheritance means we can later use or develop other methods of solving the problem.

This process is shown in Figure 6. We do not make any attempt to classify the client problem or its problem-specific classes. The solution designers can develop these classes completely independently. The main issue for the solutions designers is correctness, that is, making sure that the behavior is what the client specified. This process is outlined below.

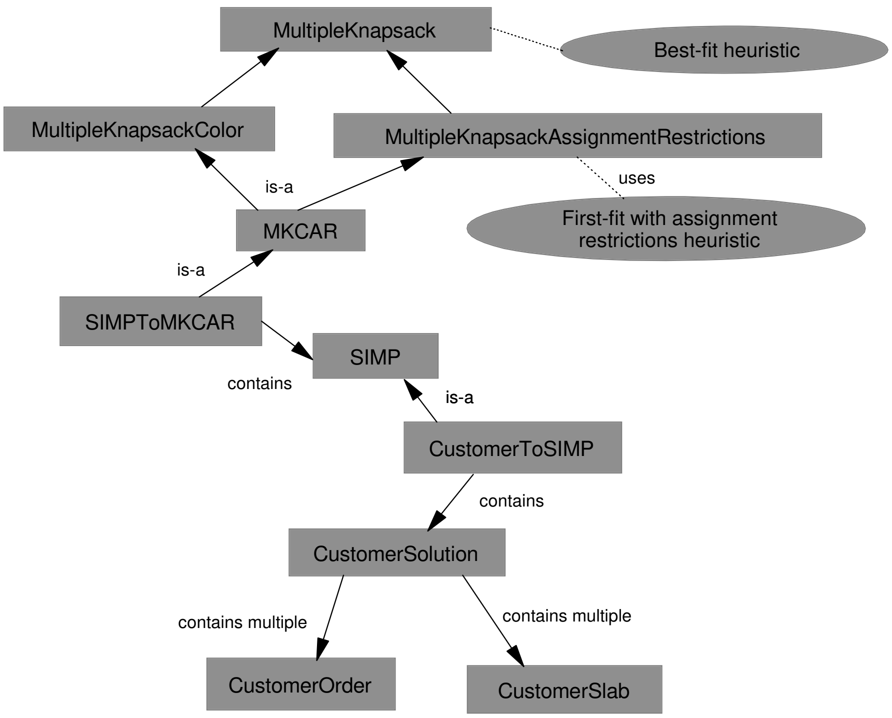

Figure 6: Deploying SIMP as an example of algorithm based software design

6 Conclusions

We have outlined a high level software design methodology for multi-project use and illustrated its use in an industrial strength case study. In this section we conclude the paper with a set of recommendations for using this methodology. These recommendations are meant to guide the use of the outlined methodology based on our experience. First we have identified a set of roles or design activities that are required for any software project. In the following subsection we provide a set of recommendations on their functions.

6.1 Partitioning Design Activities into Roles

Algorithm designers develop and refine algorithms that solve the problem/solution abstractions. The algorithm designers improve existing algorithms, add new algorithms for existing abstractions, and suggest new abstractions or updates to existing abstractions based on interactions with the solution designers and develop new algorithms for those new abstractions. They don't need to know exactly how constraints are being evaluated or the internal representation of the solution because the code they develop is written only to the virtual base classes.

Abstraction designers develop the abstractions. The abstraction designers can work with the problem designers and algorithm designers to see if the abstraction needs to be modified or a new one designed. Abstraction designers often factor new implementations into abstractions. The abstraction designers are the adapt between the solution designers and the algorithm designers.

Solution designers fit real world problems into the abstractions. The solution designers implement the (usually quite complex) constraints to capture the behavior of the problem in the form of derived classes of the abstraction virtual base classes. Solution designers implement the behavior that is hidden from the algorithm designers. They don't need to know in detail how the algorithms are going to work but should be able to converse with both the algorithm and abstraction designers.

Everybody communicates so that we have the can rapidly develop the best algorithms, abstractions, and solutions.

6.2 Recommendations

In this section we discuss some of the major insights of building software platforms in such a principled fashion and conclude with a set of recommendations for algorithm designers and for abstraction designers.

Avoid imposing data structures prematurely. For example, one way to think of a TSP problem is as a graph. However, that is only one way. The correct way to think about a TSP problem is in the language of the TSP problem: There are several **cities** each pair of which are separated

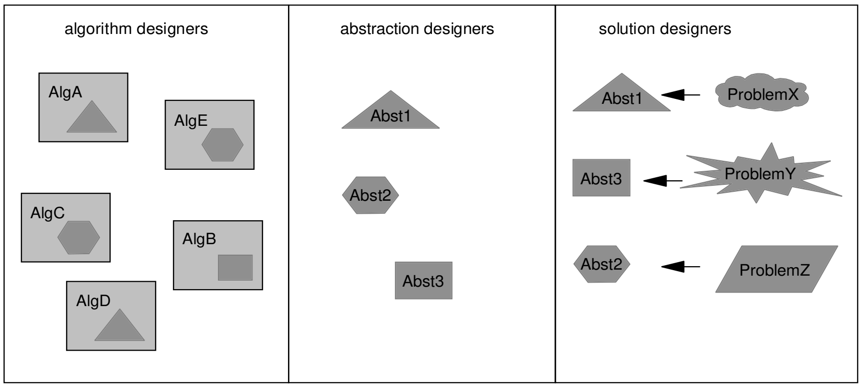

Figure 7: Software Development Roles

by a specified **distance**. You are to form a **tour** that visits every city and minimizes the total distance of the tour.

Prematurely imposing a data structure limits the flexibility to implement different solution. Admittedly, it is less convincing when talking about a TSP but consider higher level abstractions. Think of the many possible representations for a job shop scheduling problem.

The foremost consideration for an algorithm designer should be to worry less about the *representation* of the information required by the algorithm they happen to be working on and more about the *interface* to that information. Put in another way, algorithm designers should always write abstract base classes.

For example, to solve a TSP problem, one might think immediately of a distance matrix representation. However, building this data structure into the problem interface is premature because it limits the ability to derive from this object as well as forcing us to copy information frequently. Consider the case where we derive a kind of TSP problem whose graph is not complete, a SparseTSP. In this case, depending on the sparsity of the graph, a distance matrix may be a very inefficient representation. Better to leave the representation of the problem abstract and let the adapters formulate the problem as necessary.

If the object needs a constructor that takes a bunch of values and arrays of values, something is probably wrong. Following this rule goes a long toward eliminating copying. The client problem will simply provide the implementation to the interface required by the algorithm.

Algorithm designers will likely ask, "But what about efficiency?" How can we be sure that the data will be made available in an adequately expeditious fashion? The answer is to first determine what the algorithm data access requirements are and publish them, and second leave the rest to the adapter classes. Additionally, judicious use of `const` in C++ and caching in adapter classes can mitigate any possible performance problems.

6.3 Acknowledgments*

The authors would like to acknowledge useful discussions with Richard Goodwin and Rama Akkiraju at IBM T.J. Watson Research center.

7 References

Damiani, E, Fugini, M, and Fusaschi, E,(1997) "A Descriptor-Based Approach to OO Code Reuse", Computer, October 1997, p 73ff

Etzkon, L. and Davis, C.,(1997) "Automatically Identifying Reusable OO Legacy Code", Computer, October 1997, p 66ff

Gamma E., Helm R., Johnson R., and Vlissides J.(1995), "Design Patterns : Elements of Reusable Object-Oriented Software" Addison Wesley Publishers 1995.

Jacobson, I., Griss, M., and Jonsson, P.,(1997) "Making the Reuse Business Work", Computer, October 1997, p 36ff

Kalagnanam, J., M. Dawande, M. Trumbo, and H.S. Lee (1998), "Inventory Matching Problems in the Steel industry", IBM Research Report RC 21171 (94615), April 1998.

Kalagnanam J., S. Murthy and R. J. Miller(1997) "A Workbench for Rapid Prototyping of Optimization Problems in Production Planning and Scheduling". IBM Research Report, RC 21028, July 1997.

Martello, S. and P. Toth (1989), "Knapsack Problems", John Wiler and Sons, New York (1989).

Mehlorn, K., S. Naher, and C. Uhrig (1997), "The LEDA User's Manual", Max Planck Institute, Saarbrucken, Germany (1997).

A The Surplus Inventory Matching Problem Class

```
class SIMP
{
public:

  // Minimum and maximum weight per unit of this order

  virtual double unitMinWeight(int orderId) const = 0;
  virtual double unitMaxWeight(int orderId) const = 0;

  // Minimum, target, and maximum weight for this order
```

```
virtual double orderMinWeight(int orderId) const = 0;
virtual double orderTargetWeight(int orderId) const = 0;
virtual double orderMaxWeight(int orderId) const = 0;

// By default, the order is filled if it has it´s minimum weight applied

virtual int orderIsFilled(int orderId) const;

// The current minimum and maximum weight for a slab.

virtual double slabMaxWeight(int slabId) const = 0;
virtual double slabMinWeight(int slabId) const = 0;

// Iterate over all possible matches for this order and slab

IAIterator<int> orderMatchIterator(int orderId) const = 0;
IAIterator<int> slabMatchIterator(int slabId) const = 0;

// Return the match corresponding to the specified unit

int unitMatch(int unitId) const = 0;

// Return the order and slab corresponding to the specified match

int matchOrder(int matchId) const = 0
int matchSlab(int matchId) const = 0;

// The current dimensions of a slab.

virtual double slabWidth(int slabId) const = 0;
virtual double slabLength(int slabId) const = 0;
virtual double slabThickness(int slabId) const = 0;
virtual double slabDensity(int slabId) const = 0;

// By default the weight is width*thickness*length*density

virtual double slabWeight(int slabId) const;

// The weight of orders on a slab. By default, the sum of the
// product of unit weight and unit count for all units.

virtual double slabAppliedWeight(int slabId) const;

// Test how much of the order can be filled by the slab. The unit
// size and unit count returned represent what the new application
// would be, not the delta. For now, let´s also say that canApply
// returns a handle that must be passed to doApply. As before, if
// a non-zero handle is received, the application is not feasible.

virtual int canApply(int match, double & unitSize, int & unitCount) const = 0;

// Unapply the specified weight from the order/slab.

virtual int unApply(int unit) = 0;
```

```
  // Apply this order be applied to this slab and with the specified
  // pu size and number. This is also the mechanism whereby we change
  // the unit size/count. Once the operation is used, it is no longer
  // valid.

  virtual int doApply(int match, double unitSize, int unitCount) = 0;

  // The quantity applied for the specified order. By default, the sum
  // of the product of unit weight and unit count for all units.

  virtual double orderAppliedWeight(int orderId) const;

  // Return the number of slabs. By default use the size of the
  // iterator over the slabs.

  virtual int numberOfSlabs() const;

  // Clone this solution

  virtual SIMP * clone() const = 0;

  // Destroy this solution

  virtual ~SIMP();

  // Iterate over the slabs

  virtual IAIterator< int > slabIterator() const = 0;

  // Iterate over the orders

  virtual IAIterator< int > orderIterator() const = 0;

  // Iterate over the units of an order

  virtual IAIterator< int > orderUnitIterator(int orderId) const = 0;

  // Iterate over the units of an slab

  virtual IAIterator< int > slabUnitIterator(int slabId) const = 0;

  // Return the unit weight of the specified unit

  virtual double unitWeight(int unit) const = 0;

  // Return the unit count of the specified unit

  virtual double unitCount(int unit) const = 0;

  // The weight of the units applied. By default the product of the
  // unit count and the unit weight.

  virtual double unitAppliedWeight(int unit) const;

};
```