Software Reuse With a Library of Abstractions

Jonathan Phillips

September 26, 2003

Overview

- Overview of reuse and motivation
- Problem Formulation
- Solution Approach
- Solution Example

What is Software Reuse?

- Using existing software artifacts during the construction of a new software system.
  - Includes specifications, designs, code, documentation, test cases, etc.
- Manual Reuse
- Automated Reuse

Manual Reuse

- Code scavenging
- Components
- Schemas
- Architectures/Patterns

Motivations

- Redundancy
- Cognitive distance, efficiency
- Cost
- Systematicity
- Reliability

Automated Reuse

- NGL (Nth Generation Language)
- Application generators
  - Domain specific
- Transformational systems
- Component Retrieval

Motivations

- Abstraction
- Specification in the large
- Speed, cost
- Testing
  - Make different flavors
- Formalize domains

Problem Formulation

"To reuse old software in the generation of a new system." This problem will have three sub-problems that need to be addressed.

- Specification
- Component Reuse
- Component Retrieval

Problem Formulation

Given a user defined problem P such that:

P = <Sp, R, A> where

Sp is a set of formal specifications $(sp_1, sp_2, \ldots)$,

A is a set of abstractions used to help define and augment the specifications and relationships $(a_1, a_2, \ldots)$, and

R is a set of relationships between the specifications and/or abstractions, $(r_1, r_2, \ldots)$ where r_i is a tuple relating two or more specifications/abstractions and a relationship type

Generate code that satisfies the specifications, relationships, and abstractions found in P.

Specification Example : Eight Puzzle

- Informal Spec: Find a solution to an eight puzzle using A* search.
- Using the described formalism:
  - We need 2 formal specifications, 1 describing the eight puzzle, 1 describing a heuristic needed for the A* search
  - Abstractions describing the major elements of the system
  - Relationships between these abstractions and the formal specifications.

Formal Specifications

Eight Puzzle
instance variables:
currentState array size 9
currentState[i] $\in$ {1, 2, …., 9} $\forall$j, k currentState[j] $\neq$ currentState[k]
goalState array size 9
goalState[i] $\in$ {1, 2, …., 9} $\forall$j, k goalState[j] $\neq$ goalState[k]
lastOperator int $\in$ {0, 1, 2, 3}

operations:
up return Eight Puzzle e s.t.
e.currentState.indexOf(9) = self.currentState.indexOf(9) – 3
and e.currentState.indexOf(9) + 3 = self.currentState.indexOf(9)
and e.lastOperator = 0
if self.currentState.indexOf(9) > 2

down …
left …
right …

ManhattanHeuristic
instance variables:
goal array size 9 of int contents 1…9

operations:
distance(input array size 9 of int contents 1…9)
return d = $d_1 + d_2 + \ldots + d_9$
d_i = abs((goal.indexOf(i) mod 3) – (input.indexOf(i) mod 3)) +
abs((goal.indexOf(i) / 3) – (input.indexOf(i) / 3))

Relationships and Abstractions

- Abstractions:
  - FindSolution(EightPuzzle)
  - Heuristic(ManhattanHeuristic)
- Relationship:
  - Uses(FindSolution, Heuristic)

More Formally

A specification abstraction is:

$A_s = <A_n, Param>$

A_n is the name of the specification abstraction

Param is the set of parameters. Each element of param must be present in P as a formal specification or abstraction

A specification relationship is similar to a specification abstraction:

$R_S = <R_n, Param>$

R_n is the name of the relationship

Param is the set of abstractions and/or specifications involved in the relationship

A relationship is a specialized type of abstraction and included here to aid in understanding.

Problem Formulation: Automated Component Reuse

- Need to identify all components necessary to build a solution based on the input specification.

Let:

C_L be the library of components

P be the problem specification

Find the set of components:

C_{sol} $\forall c_i \in C_{sol}$, $c_i \in C_L$

such that they can be composed into solution S where S solves P.

Ex. The eight puzzle solution will consist of the following components: EightPuzzleProblem, MinPriorityQueue, AStarSearch, ManhattanHeuristic, EightPuzzleProblemSolver, and OrderedNode(not discussed here)

Problem Formulation: Component Retrieval

- Maintain a component library that can return components based on user queries and matching criteria.
- If a component cannot be found to satisfy some portion of P (Ex. EightPuzzle or ManhattanHeuristic) then the user should be able to specify a component such that the component fits into an associated abstraction and meets all the requirements of that abstraction.

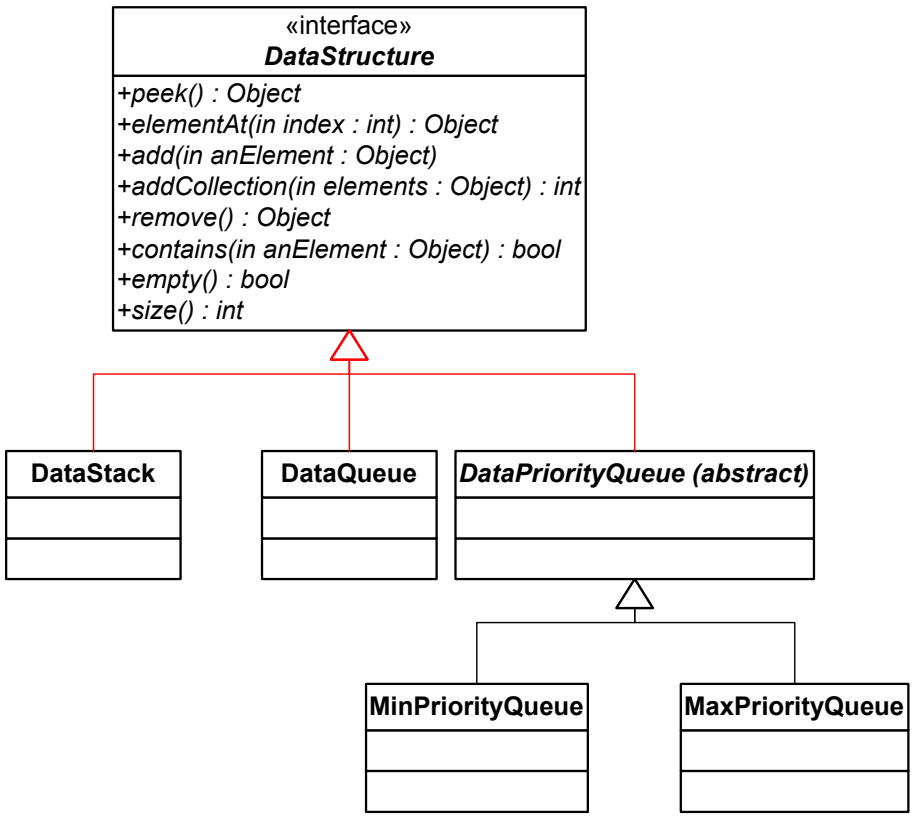
Solution Approach

- Use a library of abstractions
  - In specification
  - Guide solution construction
- Solutions built through automated concretization of abstract specifications.
- Reference components using abstractions from library.

Abstraction Library

- Hierarchical
- General
- Domain specific

Abstraction Examples: Class Hierarchy



- Types of abstraction involved in this class diagram
  - Interface
  - Abstract Class

  } Java Abstractions
  - Inheritance →
  - Implementation →

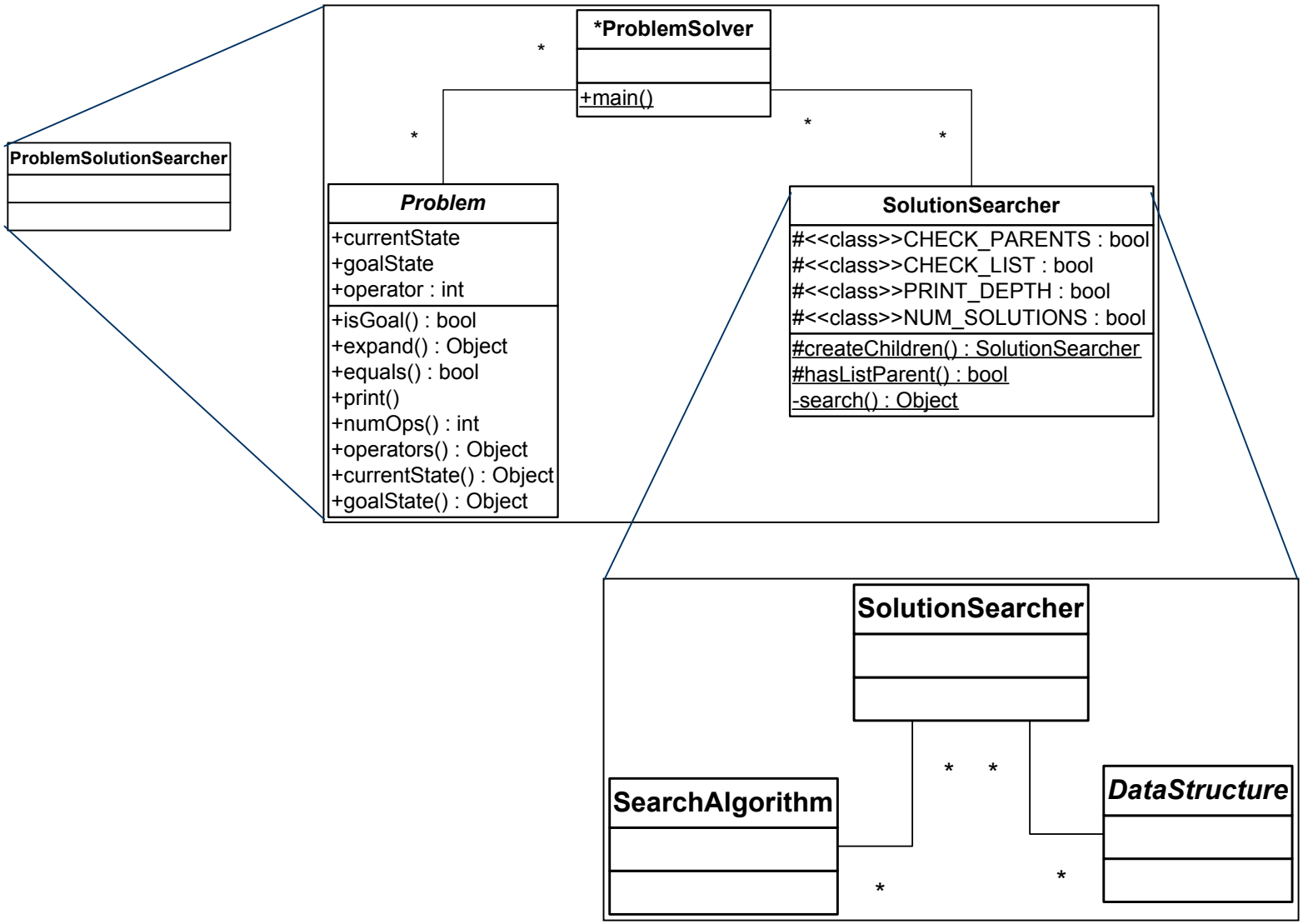
Hierarchical Abstractions

- Specializations
- Implementations
- Decompositions

Hierarchical Decompositions

- Problem solving domain:
- Problem Solver consists of:
  - Problem specification (abstract class)
  - Solution searching component
    - Data Structure
    - Searching algorithm

Decomposition Examples



General Vs. Specific

- General abstractions semantics
  - Apply to the abstraction wherever it is seen
- Specific abstraction semantics
  - Only specific components' abstractions
- Determine how abstractions are implemented
- Abstraction as classes vs. instances
- Missing info filled in by general abstraction

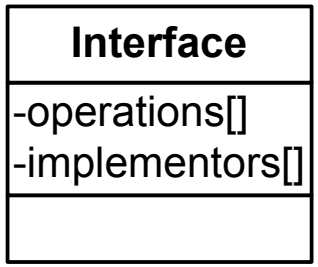
General V. Specific Example

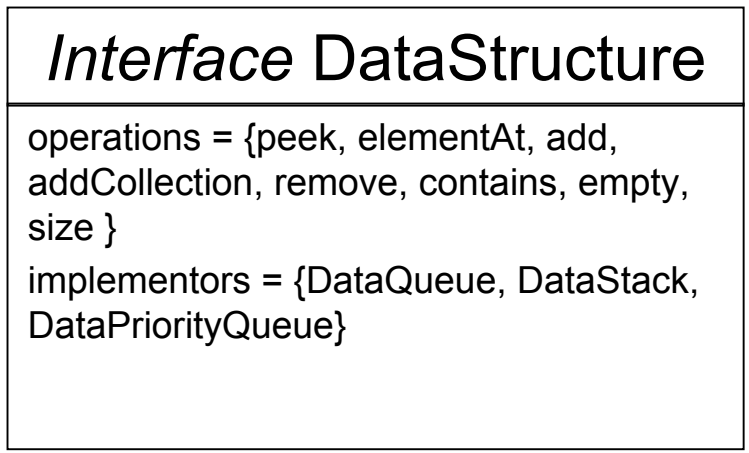
Interface Abstraction

- Slots for operations
- Slots for implementing abstractions

DataStructure Interface

- Contains actual operations and implementing classes





Domain Specific Abstractions

- Define operations, components, classes, relationships in a domain
- Result of domain analysis
- Implementations of General abstractions
- Often Hierarchical (see previous example)
- Ex: Problem, SolutionSearcher, DataStructure

Reminder:
Abstractions Throughout the Solution

- Specification
- Component Retrieval
- Concretization guides.

Abstractions in Specification

- Form a specification language
- Simplify specification
- Speeds solution generation
- Quickly identify “holes”
- Building blocks for rest of solution algorithm

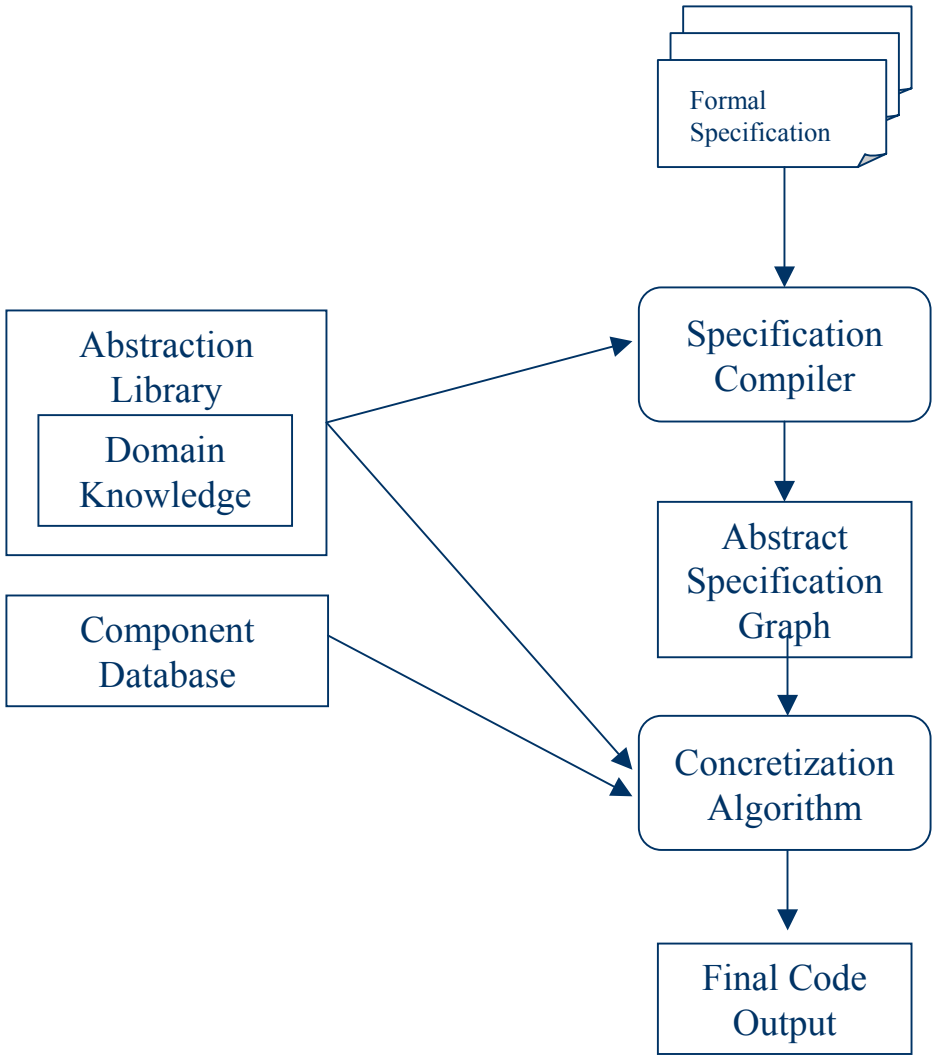
Component Retrieval

- Positives of this approach
  - Easy when pointed to by an abstraction (DataStructure only has 4 choices!)
  - Already given a framework to fit into
  - Framework possibly designed for the component
- Negatives
  - Harder to use components outside original scope
  - Ungainly to include additional “normal” retrieval system

Towards Concretization: How Abstractions are Used Along the Way

- System starts with specification that uses abstractions
- This specification is refined so abstractions are more formal and match well with library abstractions (improve on what the user has given)
- Concretize the abstractions based on their own structures and natures
  - Decompose hierarchical abstractions
  - Select abstractions that implement interfaces, abstract classes, or roots of class hierarchies
- Retrieve components when no abstractions remain

System Overview



Specification Compiler

1. Find any abstractions within the specification that the user did not explicitly determine and combine them with the given abstractions and relationships.
2. Refine the abstractions and relationships
3. Return a “graph” of abstractions that is ready for processing in the concretization/instantiation system.

Abstract Specification Graph

- Traversing the graph and concretizing nodes (if possible) builds the solution
- Graph edges show how sub-systems of the solution interact

Abstract Specification Graph Ex.

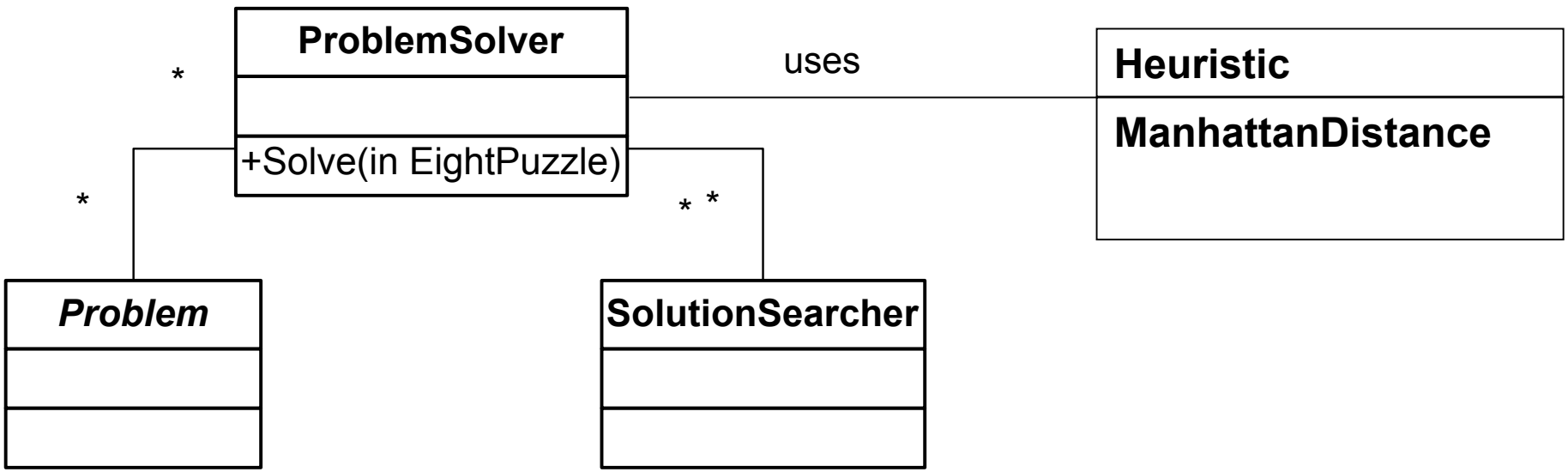
Ta Daaa!!!

Concretization Algorithm

- Select the next abstraction to concretize
- Concretize it by:
  - Specializing it, choosing an implementation, or expanding it
- Add any new abstractions back into the graph in place of the former abstraction

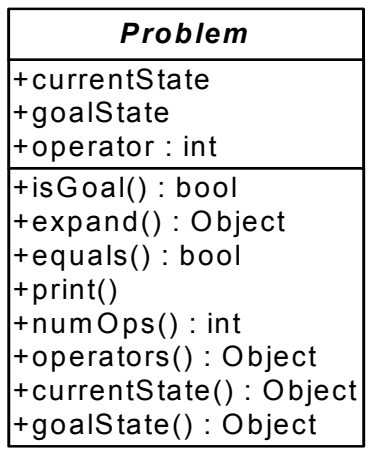
Concretization Ex.

- Begin by concretizing the ProblemSolutionSearcher. The decomposition shown has been seen previously. The algorithm adds the new abstractions to the graph and chooses the next one to process.
- In this case it is the ProblemSolver



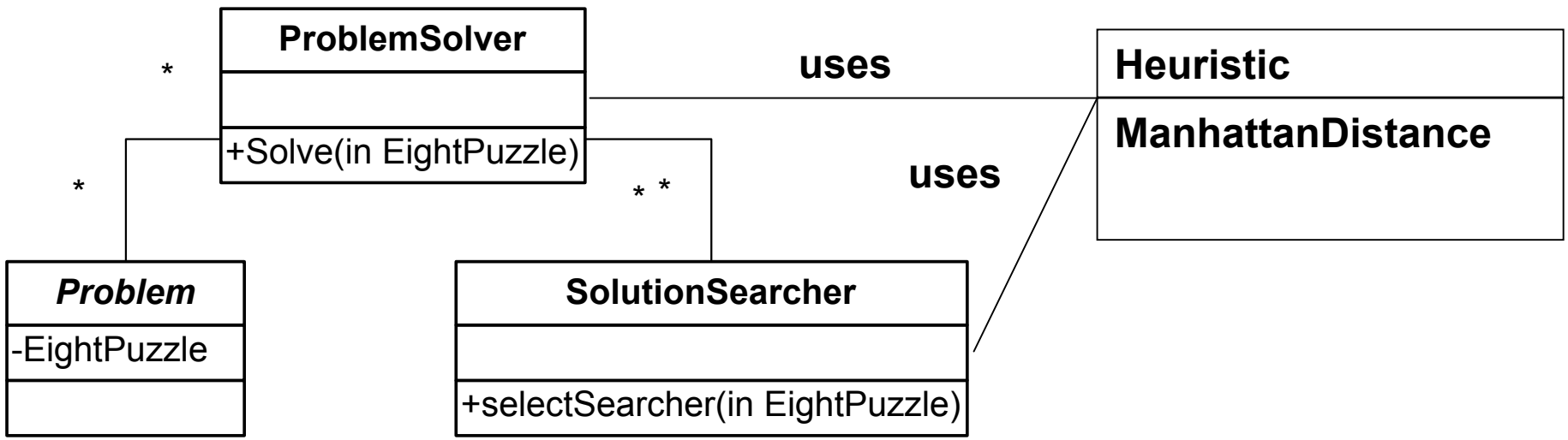
Problem Solver and Problem

- Knows its children (Problem, SolutionSearcher)
- Has internal order of concretization
- Gives Problem as the next abstraction to concretize in the solution algorithm
- Problem is an abstract class
- Looks among its implementations for a class that contains the correct operators obtained from EightPuzzle
- Assuming none is found
  - User fills in the missing code
  - Automatically generate the code based on the formal specification and Problem abstraction



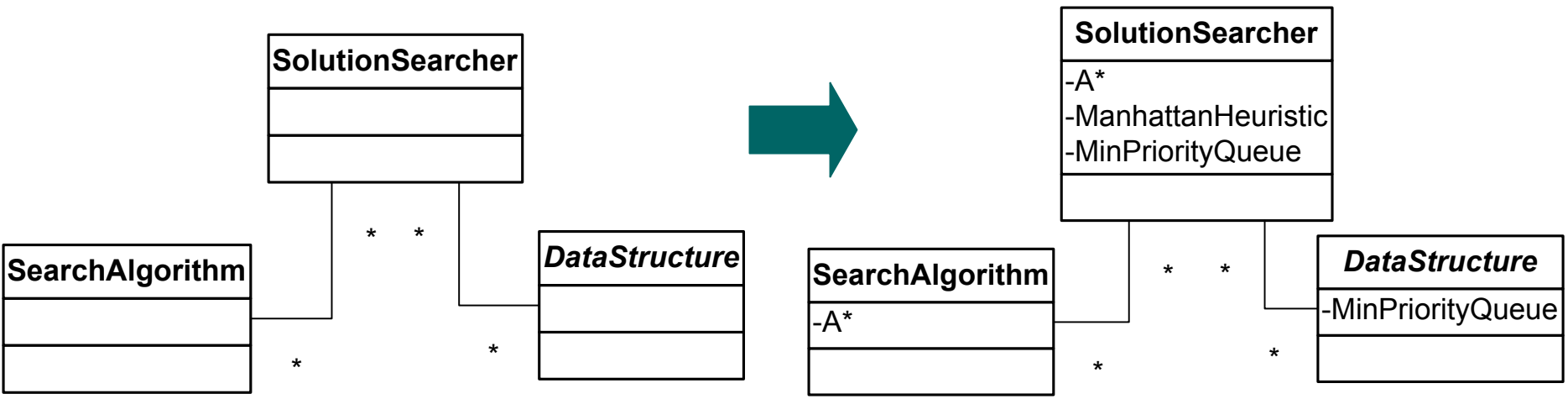
ProblemSolver and SolutionSearcher

- Problem didn't need to know Heuristic, but SolutionSearcher does.
- ProblemSolver passes the uses relation down to SolutionSearcher to allow it to decide to use it or not



SolutionSearcher and Sons

- The algorithm selected dictates the data structure and the algorithm is influenced by the heuristic.
- The final state after concretization of this abstraction shows that an A* search was selected using the Manhattan heuristic and that a MinPriorityQueue will be used to hold expanded nodes in the search.



Finishing ProblemSolver

- ProblemSolver knows the problem to use and the search algorithm to use and since A* "uses" the heuristic, the relationship is satisfied.
- How is the solution composed?
  - Problem and SolutionSearcher are black boxes
  - ProblemSolver must change based on the results returned from its children

ProblemSolver code.

```
import java.io.*;
import java.util.*;

public class EightPuzzleProblemSolver {
	public static void main(String[] args)
	{
		EightPuzzleProblem problem = EightPuzzleProblem.PromptForEightPuzzleProblem(System.in);
		ManhattanHeuristic heuristic = new ManhattanHeuristic(problem.goalState());
		Vector solutions = HeuristicSolutionSearcher.AStarSearch((EightPuzzleProblem)problem, heuristic);

		if (solutions.size() > 0) {
			int[] nextSolution;
			System.out.println();
			for (int i = 0; i < solutions.size(); i++) {
				nextSolution = (int[])solutions.elementAt(i);
				for (int j = 0; j < nextSolution.length; j++)
					System.out.print(nextSolution[j] + " ");
				System.out.println();
			}
		}
		else {
			System.out.println("\nNo solutions to this problem exist");
			System.out.println();
		}
	}
}
```

Conclusions

- We've only seen a few abstractions
  - Design patterns
  - More domains
- Need a formal abstraction methodology (abstract syntax?)
- Formal code integration
- Program Synthesis?

¿Preguntas?