From Problems to Programs

A Pattern Language to Go from Problem Requirements to Solution Schemas in Elementary Programming [*]

José Manuel Burgos-Ortiz
Javier Galve-Francés Julio García-Martín

{jmburgos,jgalve,juliog}@fi.upm.es

Universidad Politécnica de Madrid[**]

Motivation

The *From Problems to Programs* pattern language is an intent to formalize to a certain extent, via patterns, the programming process to go from the problem to the program. It outlines a common instructional framework to categorize, specify and solve the problems commonly taught in an introductory programming course. The language helps those educators interested on an organization of information for teaching a programming course which puts primary interest on the structure of and relationships between a problem, a program and a solution schema. The language can also help to see how formal specification concepts are related with programming.

In the usual "syntax-driven" approach to programming instruction, problems are introduced as examples needed to explain a syntactic feature and solutions are presented as completed programs that solve the problems. Thus this collection of unrelated problems and last-version programs is the material that students handle to solve new problems. This course organization does not promote knowledge integration [2] as it does not show how problems are related to each other, neither how problems are related to programming language constructs [6] nor how is the "continuum" to go from the problem to the program along the programming process. A very small deal of attention is given to the skills that an expert programmer handles to develop a program [7], starting from the problem as goals and going to the solution through intermediate abstractions -differently named as canned solutions, schemas [4] or plans [7]- to represent program solutions.

Some authors have underlined a fact observed in research about knowledge acquisition: a good organization of knowledge is the key for the effectiveness of instruction. A taxonomy of problems lets stablish families of problems, very useful to find solutions to new problems by searching analogies with previously solved ones. The pattern language *From Problems to Programs* relies on a proposed taxonomy of problems. This taxonomy lets talk about the "chunks" of programming knowledge as abstract solution schemas that are based on the classification of the problem at the analysis stage. The language provides a guide to go from a problem statement to a solution schema, starting

[*] This work has been partially supported by the Spanish project F.G.UPM-43700000190.
[**] Department of LSIIS, Technical University of Madrid, Boadilla del Monte, 28660 Madrid, Spain.

from a first formulation of the problem and going layer by layer along the development process, refining the formulation until one of the solution schemas is reached.

This pattern language offers educators a way to explore the effectiveness of a pattern-based approach into the elementary programming instruction, making use of patterns during the whole development process. This pattern language tries to be a speculation in use of patterns to search for comprehensive models for instructional practice, models that should be very useful for course design and for the development of new instructional tools. Of course, the taxonomy proposed is not unique, canonical or complete, but, as Soloway says in [6]: "finding exceptions and developing new structures to accommodate the inconsistencies is a powerful learning technique" and "becoming aware that abstractions admit of exceptions is precisely the kind of problem solving skill that one wants to encourage".

An Overview of the Patterns

The patterns form three groups, one for each of the first stages of the programming process: two for problem analysis (Classification and Specification patterns) and one for program synthesis (Solution patterns):

- **Classification Patterns:** A set of patterns for the requirements analysis stage used to classify programming problems. These patterns provide the first aid to analyze the problem by fixing an organized description of data and problem. The patterns are:

    1. *Data Description*: How to describe the data involved in a problem.
    2. *Problem Description*: How to describe the problem for a full understanding of it.
    3. *Categorized Problem*: How to classify the problem in the taxonomy.

- **Formal Specification Patterns:** A set of patterns to get formal specifications for problems previously classified.. They provide a rigorous description of the problem, using notation based on logic and mathematics. The specification describes the data and problem operations in an abstract manner, without any stipulations about how these should be implemented. Problems are described as functions, using PRE/POST semantics.

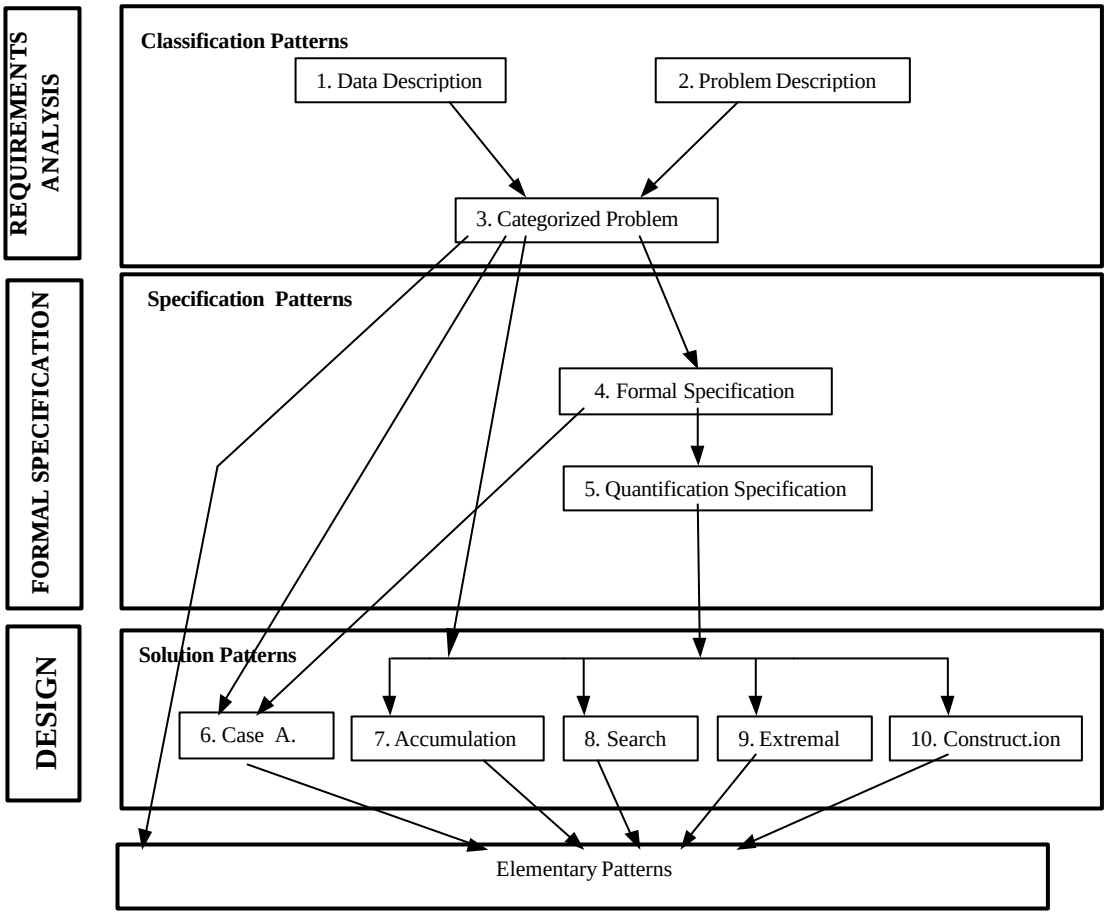
    4. *Formal Specification*: How to describe the program behaviour precisely in a formal way.
    5. *Quantification Specification*: How to describe the behaviour of a Quantification problem precisely in a formal way.

- **Solution Patterns:** A set of patterns to design solutions for problems previously specified. They allow the transformation of specifications (informal or formal) defining the effect of a program (i.e. the meaning of the problem) into an algorithmic notation (similar to a formal programming language) which defines how this effect is achieved in a procedural programming language but omitting finer details.

    6. *Case-Analysis*: How to write a solution for a Case-Analysis problem.

7. Iterative Accumulation Quantification: How to write an iterative solution for an Accumulation Quantification problem.
8. Iterative Search Quantification: How to write an iterative solution for a Search Quantification problem.
9. *Iterative Extremal Quantification*: How to write an iterative solution for an *Extremal Quantification* problem.
10. *Iterative Construction Quantification* : How to write an iterative solution for a *Construction Quantification* problem.

The use of the Specification patterns is optional. They can be bypassed, going straightly from the Classification to the Solution group: there is no need to specify a problem in order to find a solution to it. The Solution patterns connects with the patterns of the Elementary Patterns Group [8], which provide the necessary details to implement the solutions. In order to go from the solution patterns to the elementary patterns, a step of interptetation is necessary and to go reversely from the elementary stuff to the solution patterns, a step of abstraction should be taken.



Structure of *From Problems to Programs*

How to Use The Patterns

Starting from the statement of the problem, it is defined (as a function) with a premise (precondition), and a goal (postcondition) with the *Problem Description* pattern. The premise is often known as precondition and the goal as postcondition. The data involved in the problem - inputs and outputs - must be described as well as their respective domains (with the *Data Description* pattern). Once we have the set of inputs, outputs, premise and goal, we can classify the problem into the taxonomy by using the *Categorized Problem* pattern. As English can be ambiguous, the specification may rely (but not necessarily) on a formal specification language. The *Formal Specification* pattern helps in doing this. With the problem categorized, the specific solution schema for that category can be used, applying one of the solution patterns: *Case-Analysis*, or one of the *Iterative Quantification* patterns: *Accumulation, Search, Extremal* or *Construction.*

The Patterns

1 Data Description

(Also Known as: *Data Specification.*)

You want to write a program that solves a problem described as a requirements statement written -or spoken- in English. By using *Problem Description* you are describing what to do with the "things" you find in the problem. Now you want to describe those "things", that we name as *data objects* or simply *problem data.*

The problem statement provides an imprecise, ambiguous and often incomplete description of the data involved in the problem. It is initially put forward (written or spoken) by somebody who may have not considered the problem deeply enough and/or may have not a sufficient appreciation of computer programming to be aware of the absolute exactness needed. The data may be complex and there may be many mixed details about different data. Some constraints on the data are imposed by the problem.

Therefore seek for all the data domains to which the problem data belong. Every data object belongs to a generic category of data, *data type* or *data domain* (since a type is a set of values, both terms will be used indistinctly) and there may be many data objects that belong to the same data type. Choose good names to identify the data domains. The data structure (representation details of a data object in a program) have not to be considered at this analysis stage. Give an informal description of data domains as texts (written in English) enclosed in quotation marks, including, if there is any, the restrictions.

Use the usual predefined domain names (Natural, Integer, Real, Char, Boolean, etc…). Define complex data as aggregations of data (heterogeneous items), collections of data (homogeneous items) or combinations of both. Use explicitly in the data description the statements "Aggregation of .." and "Collection of ..". Use the "= ..." expression to define data domains by renaming, if necessary. The data types may be hierarchies of other data types. Detect these hierarchies and reflect them mentioning the names of the types on which more complex types are based. These hierarchies will guide the design of the program.

Suppose the problem statement: *"Given a collection of regular polygons, return a collection of their areas"*. The data of this problem can be modelled as:

```
TPoint = "Aggregation of two objects of the type Real"
TPolygon = "Collection of at least three objects of the type TPoint"
TPolygons = "Collection of objects of the type TPolygon"
TArea = Real
TAreas = "Collection of objects of the type TArea"
```

Now the data in your problem have been described and when you have your problem described as well, you will be in a position to categorize it.

2 Problem Description

(Also Known as: *Specification, Problem Specification, Understanding the Problem, Understanding the Program's Purpose*.)

You want to write a program that solves a problem described as a requirements statement written -or spoken- in English. You want to describe what to do with the "things" you find in your problem.

The problem statement provides an imprecise, ambiguous and often incomplete description of the problem. It is initially put forward (written or spoken) by somebody who may have not considered the problem deeply enough and/or may have not a sufficient appreciation of computer programming to be aware of the absolute exactness needed. The problem statement may be complex with many mixed details and may even be composed of several problem statements. The problem data may impose some restrictions on the problem about its definition scope (i.e., the values of the data for which it is defined) that have to be assumed before solving it.

Conceptually, a program is equivalent to a function in mathematics. The set of all possible input and output items to and from the program is respectively equivalent to the input and output domain of the function.

Therefore give the program a meaningful name and re-write the problem statement (and the subsequent subproblems) as a function using the PRE/POST or *contract* model, attaching previously the data domain descriptions D and R obtained from the *Data Description* pattern, as follows:

```
D = "Input domain description"
R = "Output domain description"
FUNCTION AProblem: (x : D) -> (y : R)
PRE: "Precondition description"
POST: "Postcondition description"
FUNCTION ASubProblem: ..
        ...
```

Give meaningful names to the input data (x) and output data (y). These variables are also known as program´s parameters. Write informal statements to describe the problem's pre- and post-condition (the preconditions are assumptions made about the input data; they describe the values for which the problem is not defined. The post-condition describes the intended meaning of the problem, as a relationship between the input data and the output data). Use "true" as the pre-condition for a problem with no constraints and when several different pre-conditions are found, combine them using the "*and*" logic operator. When the problem statement is too complex, do not describe its post-condition in only one hit: split the problem into simpler sub-problems (piecemeal description), then apply the *Categorized Problem* pattern to each of the sub-problems in isolation and, finally, combine their post-conditions. Attach the description of the subproblems at the main problem's description.

Suppose the problem statement: "*Given a collection of non-empty strings, check if at least one of them includes, in its upper case letters, a given upper case letter*". Applying the *Data Description* pattern, the problem data are:

```
TString = "Collection of Char"
```

```
TStringCollection = "Collection of TString"
```

There is a subproblem described as: *"Check if an upper case letter is present at least once in the upper case letters of a non-empty string"* and modelled as follows:

```
FUNCTION IsPresent: (c : Char) x (string : TString) -> (isPresent : Bool)
PRE: "string is not empty" and "c is an upper case letter"
POST: isPresent = "c is present in the upper case letters of string"
```

The main problem can be described now as: *"Given a collection of non-empty strings and an upper case letter c, check if* `IsPresent (c, string)`*"*, and modelled as:

```
FUNCTION IsStringPresent: (c : Char)x(collection: TStringCollection)-> (result: Bool)
PRE: "All strings in collection are non-empty strings"and "c is an upper case letter"
POST: result="There is a string in collection such that IsPresent(c,string)"
```

Now your problem has been described and when you have your data described as well, you will be in a position to categorize it.

3 Categorized Problem

You have both a problem and a data description for your problem statement.

A problem description from the scratch is hard and may yield a bad description. Most of the problems are related in some way to other problems by some kind of affinities that may not appear on the surface and should be found out.

Therefore consider using the POST section in the *Problem Description* pattern and classify the problem by pigeonholing it in one of the categories listed on the table below. It may be helpful to rewrite it using the most similar terms. Attach the category name at the end of the POST description, in brackets.

Category		POST	Functionality
Direct Solution		The result is a simple expression or is obtained from a previously solved problem or is a composition of them.	-
Case Analysis		The result takes different values depending on input data.	-
Quantification	Sum	Sum an expression applied on every item in a collection.	Accumulation
	Times	Multiply an expression applied on every item in a collection.	
	Count	Calculate how many items in a collection verify a property.	
	Exists	Check if at least one item in a collection verifies a property.	Search
	Search	Search for an item in a collection that verifies a property.	
	ForAll	Check if each item in a collection verifies a property.	
	Max	Calculate the maximum value of an expression in a collection of items.	Extremal
	Min	Calculate the minimum value of an expression in a collection of items.	
	Map	Construct a new collection as the result of applying an expression to each element of a collection.	Construction

If the collection of data is determined by a condition that the input data have to verify, the problem formulation will be, for instance, for the Sum quantification something like

"*Sum an expression applied on every item in a collection that verifies a property*". The property is known as the *filter*.

If the problem does not match one of the above categories, then reformulate the description (the POST section) in a different way.

Suppose the problem statement: "*Check if an upper case letter is present at least once in the upper case letters of a non-empty string*" modelled as:

```
FUNCTION IsPresent: (c : Char) x (string : TString) -> (isPresent : Bool)
PRE: "string is not empty" and "c is an upper case letter"
POST: isPresent ="c is present in the upper case letters of string"
```

The problem belongs to the *Exists* category, because its POST description is equivalent to

> "*Check if at least one character in the upper case letters of string verifies that is equal to c*".

So that it can be refined to

```
FUNCTION IsPresent: (c : Char) x (string : TString)-> (isPresent : Bool)
PRE: "string is not empty" and "c is an upper case letter"
POST: isPresent =  "Check if at least one character in the characters of string
                    such that are upper case letters verifies that is equal to c"
                    (Exists)
```

Consider another problem statement: "*Determine how many times a character c is present in a string*". It can be straightly matched with the *Count* category.

```
FUNCTION NumberOfTimes: (c : Char) x (string : TString) -> (numberOfTimes : Nat)
PRE: true
POST: numberOfTimes  = "Determine how many times c is present in string" (Count)
```

Now your problem is correctly categorized and you are in a position to write a full specification of it.

4 Formal Specification

(Also known as: Formal Problem Specification).

You have the problem described informally and categorized. The problem data are also described. You want to write a formal specification for your problem.

An informal description of a problem may be ambiguous, incomplete and imprecise, probably hiding some relevant details. Informal descriptions do not always aid to detect similarities. A high level of abstraction hides relevant information for problem analysis, but a too low level of abstraction may make the problem incomprehensible. The problem description does not provide details about the order in which operations have to

be achieved neither about the restrictions on them. Sometimes, new problem's constraints appear when it is described precisely.

Therefore specify the meaning of the problem rewriting both the pre- and post-condition as logic predicates using conventional logic and mathematical notation and attach them to the problem description. Write the postcondition with a specific format that depends on the category of the problem.

Specify the data using *Abstract Sequences*[1] to model collections (i.e., Seq (Nat), Seq (Seq (Integer))), tuples to model aggregations (TCartesianPoint = Real x Real), types defined by enumeration (for example, TColor = black | white | grey) and union data types (for example, TPoint = Euclid (Real x Real) | Polar (Real x Tangle)). If the data have more characteristics than can be deduced from the type, write them explicitly after the type definition as a type invariant. Examples:

```
TPoint = Real x Real
TPolygon = Seq (TPoint)
INV (pol : TPolygon) = Length (pol) >= 3
TString = Seq (Char)
TstringCollection =  Seq (TString)
```

If you have a *Direct Solution* problem, write the post-condition as *y = Exp (x)*, where *Exp* is a composition of primitive operations and previously solved functions that goes from D to R:

```
FUNCTION Aproblem: (x : D) -> (y : R)
PRE: Precondition (x)
POST: y = Exp (x)
```

If you have a *Case-Analysis* problem, write the post-condition as follows:

```
POST:
       Case1 (x) -> y = Exp1 (x) /\
                ...
       CaseN (x) -> y = ExpN (x)
```

where $Case_i$ is a predicate that takes the input data (x) and checks a condition; Exp_i is a function that goes from D to R. Ensure that the problem domain is covered by all the cases (i.e. $Case_1$ (x) \/ ... \/ $Case_N$ (x) = Precondition (x)) and that there is no case overlapping (i.e. $Case_1$ (x) /\ ... /\ $Case_N$ (x) = false).

If you have a problem categorized as a *Quantification* problem, see *Quantification Specification*.

Detect new preconditions from the restrictions imposed by any of the operations used in the postcondition, and attach them to the preconditions defined so far.

Now you have a specification for your problem and you are in a position to be guided in the solution design process.

[1] The *Abstract Sequence* operations are described in the Appendix A.

5 Quantification Specification

(Also Known as: Quantifier Specification.)

You have a problem categorized as *Quantification* problem: the result is computed from a traversal over a data domain..You want to write a formal specification for your problem.

There is a specific quantifier for each different kind of computation issued from the traversal of the data domain and each has a different specification. All the quantification specifications have a common structure and it is independent of the data.

Therefore write your problem post-condition as a quantification:

```
FUNCTION AProblem: (x : D) -> (y : R)
PRE: Precondition (x)
POST: y = Q i <- TD | Fil (x,i) . Exp (x,i)
```

where:

- *Quantifier (Q)* is a symbol or a name that represents a quantification.
- *Traversal Variable (i)* is a variable bound by the quantifier that takes values in TD.
- *Traversal Domain (TD)* is an abstract sequence that represents the data.
- *Quantification Expression (Exp)* is the expression to compute for each item of TD.
- *Quantification Filter (Fil)* is the property that each item of TD has to verify. It is optional: in some quantifications, it is subsumed in the expression.

that provides the following meaning to *Aproblem:*

> *"Assuming that the input data verify Precondition, the output data are obtained by computing repeatedly Exp on the elements of the domain TD that satisfy Filt and combining them with the operator bound by the quantifier."*

Choose a good name for the traversal variable and an appropriate traversal domain for the problem. Abstract the collection of items of input data (x) as an abstract sequence (Appendix A) .

Suppose the refined problem description:

```
FUNCTION IsPresent: (c : Char) x (string : TString)-> (isPresent : Bool)
PRE: "string is not empty" and "c is an upper case letter"
POST: isPresent = "Check if at least one character in the characters of string such
                   that are upper case letters verifies that is equal to  c"
                   (Exists)
```

It can be specified as

```
FUNCTION IsPresent: (c : Char) x (string : TString)-> (isPresent : Bool)
PRE: "string is not empty" and "c is an upper case letter"
POST: isPresent = Exists i <- [1..Length (string)] | IsUpperCase (c). string [i] = c
```

and subsuming the filter

```
FUNCTION IsPresent: (c : Char) x (string : TString)-> (isPresent : Bool)
PRE: Length (string) >= 1 /\ IsUpperCase (c)
POST: isPresent = Exists i <- [1..Length (string)] . IsUpperCase (c)/\ string [i] = c
```

You have a *Quantification* problem formally specified.

6 Case Analysis

You have used *Categorized Problem* and have your problem categorized as a *Case Analysis*. You may have specified formally your problem.

The result of your problem has to be selected depending on two or more conditions. Each result devolution is guarded by its own condition.

Therefore if you have not used Problem Specification to specify formally your problem, use *Function for Complex Condition* [3], if necessary, and use a function $Case_i$ for each condition and write a selection chunk of code using *Patterns for Selection* [1]. If you have used *Formal Specification* to specify formally your problem, write a selection chunk of code using *Patterns for Selection* with a branch of selection for each condition and writing as action the result return.

Now you have a solution for your Case-Analysis problem.

7 Iterative Accumulation Quantification

(Also Known as: *Accumulation Loop Schema*)

You have used *Categorized Problem* and have categorized it as a *Sum, Times or Count Quantification* problem. You have probably specified it with *Quantification Specification*. In your problem, you have to accumulate a function (named Exp) applied on every item in a collection that meets a property (named Fil). You are searching for an iterative solution to your problem.

The type of the result is the same as the output type of the function Exp. In order to get the result, the collection has to be fully traversed. If the collection is empty, the result of the problem is a special value bound by the specific kind of accumulation quantification: 0 for Sum and Count and 1 for Times. There is no precondition imposed on the collection of data by these Quantifications. As the searched solution is iterative, you need to use a loop to compute the result, but the kind of loop depends on the nature of the collection of data.

Therefore if your data are a numeric interval [a, b], or they are Simple Linear [5], use the *Counted Loop* pattern [5] with the following design guideline for the loop parts outlined in it (we use as name for the traversal variable *i*, for the collection of data *collection* and for the variable with the result value *result*).

Loop Part	Numeric Interval	Simple Linear
Initialization	`i := a`	`i := 0`
Loop Continuation Condition	`i <= b` [1]	`i <= (`*`Length of collection`*`) – 1` [1]

Loop Body	`if Fil (i) then`
	`  result := result ? Exp(i)`
	`fi`
		`if Fil (collection [i]) then`
		`  result := result ?`
		`  Exp (collection[i])`
		`fi`

Loop Update Statement	`i := i + 1` [1]	`i := i + 1` [1]
Post-mortem	The result is in `result`	The result is in `result`

If your data are Streamed (a file of items), Linked or traversed using an *Iterator Object* [5], use the *Conditional Loop* pattern [5] following the next design guideline for the loop parts outlined in it (the names chosen for the traversal variable is *i*, for the collection of data is *collection*, for the iterator object is *iter* and for the variable with the result value is *result*, but any other names that follow the guidelines in the *MeaningfulVariable* pattern [3] are valid).

Loop Part	Streamed	Linked	Iterator Based
Initialization	`Reset (collection)`	`p := collection`	`iter:=collection.Iterator`
Loop Continuation Condition	`not EOF (collection)`	`p <> NIL`	`not iter.IsDone`

Loop Body	`if Fil (CurrentItem (collection)) then`
	`  result := result ?`
	`  Exp (CurrentItem (collection))`
	`Fi`
		`if Fil (p^item) then`
		`  result := result ?`
		`  Exp(p^item)`
		`fi`
			`item := iter.Next;`
			`if Fil (item) then`
			`  result := result ?`
			`  Exp (item)`
			`fi`

Loop Update Statement	`colection := Get (collection)`	`p := p^.next`	-
Post-mortem	The result is in `result`	The result is in `result`	The result is in `result`

Now you have an iterative solution for your problem.

8 Iterative Search Quantification

(Also Known as: *Search Loop Schema*)

You have used *Categorized Problem* and have categorized it as an *Exists, ForAll or LinearSearch Quantification* problem. You have probably specified it with *Quantification Specification*. In your problem, you have either to search whether there exists an item in a collection of data that meets a property named Exp (*Exists*) or whether there exists an item that does not meet a property (*ForAll*) or search for an item that meets the property (*LinearSearch*). You are looking for an iterative solution to your problem.

[1] Implicit in Counted Loop in some programming languages.

The result is a boolean for the *Exists* and *ForAll* quantifications and is of the same type as the items in the collection in the case of the *LinearSearch* quantification. In order to get the result, the collection does not always need to be fully traversed. The collection may only be empty if the problem is an *Exists* or a *ForAll* quantification. In such cases, the result of the problem is a special value bound by the specific kind of search quantification: false for *Exists* and true for *ForAll.* The problem may impose a filtering condition on the collection of data.

There is no precondition imposed on the collection of data by these quantifiers. If the problem is a *LinearSearch*, the collection of data has to meet as precondition not to be empty. As the intended solution is iterative, you need to use a loop to compute the result, but the structure of the loop depends on the nature of the collection of data and on the specific quantification.

Therefore use a *Counted Loop* pattern [5] with a local boolean variable in the Loop Continuation Condition, following different design guidelines depending on the specific quantification and the nature of the collection of data (depending on they are a numeric interval [a, b], or *Simple Linear* [5], or *Streamed* (a file of items), *Linked* or traversed with an *Iterator Object* [5]).

If there is a filtering condition (*filter*) for the collection of data, subsume it in the property *Exp* combining the filter with the property by means of the logic operator "and".

If your problem was categorized as an Exists problem, follow the next design guideline for the loop parts outlined in the *Conditional Loop* pattern [5] (the names chosen for the local variable is *found*, for the traversal variable *i*, for the collection of data *collection* and for the variable with the result value *result*, but any other names that follow the guidelines in the *MeaningfulVariable* pattern [3] are valid).

Loop Part	Numeric Interval and Simple Linear	Streamed, Linked and Iterator Based

Initialization	Numeric Interval:
	`i := a;`
	`found := false;`
	
	Simple Linear:
	`i := 0;`
	`found := false;`
		Streamed:
		`Reset (collection);`
		`found := false;`
		
		Linked:
		`p := collection;`
		`found := false;`
		
		Iterator Based:
		`iter := collection.Iterator;`
		`found := false;`

Loop Continuation Condition	Numeric Interval:
	`i <= b`
	
	Simple Linear:
	`i <= (`*`Length of collection`*`) – 1`
		Streamed:
		`not EOF (collection) and not found`
		
		Linked:
		`p <> NIL and not found`
		
		Iterator Based:
		`not iter.IsDone and not found`

Loop Body	Numeric Interval:
	`if Exp (i) then`
	`found := true`
		Streamed:
		`if Exp (CurrentItem (collection)) then`

	else
	  i := i + 1
	fi
	
	Simple Linear:
	if Exp (collection [i]) then
	  found := true
	else
	  i := i + 1
	fi
		found := true
		else
		  collection := Get (collection)
		fi
		
		Linked:
		if Exp (p^.item) then
		  found := true
		else
		  p := p^.next
		fi
		
		Iterator Based:
		item := iter.Next;
		if Exp (item) then
		  found := true
		fi

Loop Update Statement	-	-
Post-mortem	The result is in the local variable `found`	The result is in the local variable `found`

If your problem was categorized as a *Search* problem, the design guideline is the same as for the *Exists* quantification, except the Post-mortem part, outlined as follows.

Loop Part	Numeric Interval and Simple Linear	Streamed, Linked and Iterator Based
Post-mortem	Numeric Interval: The result is in the local variable `i`. Simple Linear: The result is `collection [i]`.	Streamed: The result is `CurrentItem(collection)`. Linked: The result is `p^.item`. Iterator Based: The result is `item`.

If your problem was categorized as a *ForAll* problem, follow the same design guideline as for the *Exists* quantification, using the negation of the property Exp and the computed result will be the negation of the local variable `found`.

Now you have an iterative solution for your problem.

9 Iterative Extremal Quantification

(Also Known as: *Max-Min Loop Schema*)

You have used *Categorized Problem* and have categorized it as a *Max, or Min Quantification* problem. You have probably specified it with *Quantification Specification*. In your problem, you have to determine the maximum (Max) or minimum (Min) value of a function (named Exp) among the items in a disordered collection. You are searching for an iterative solution to your problem.

The type of the result is the same as the output type of the function Exp. In order to get the result, the collection has to be fully traversed. The collection of data has to meet as precondition not to be empty. The maximum and minimum can be calculated for any type which defines an ordering relationship. As the searched solution is iterative, you need to use a loop to compute the result.

Therefore use the *Counted Loop* pattern [5] following the next design guideline for the loop parts outlined in it (the names chosen for the traversal variable is *i*, for the collection of data is *collection*, for the iterator object is *iter* and for the variable with the result value is *result*, but any other names that follow the guidelines in the *MeaningfulVariable* pattern [3] are valid).

Loop Part	Numeric Interval	Simple Linear
Initialization	`i := a + 1;` `result := Exp (a);`	`i := 1;` `result := Exp (collection [0])`
Loop Continuation Condition	`i <= b` [1]	`i <= (`*`Length of collection`*`) – 1` [1]

Loop Body	`if Exp (i) >`[2] `result then`
	`result := Exp (i)`
	`fi`
		`if Exp (collection [i]) >` [2] `result then`
		`result := Exp (collection[i])`
		`fi`

Loop Update Statement	`i := i + 1` [1]	`i := i + 1` [1]
Post-mortem	The result is in `result`	The result is in `result`

Loop Part	Streamed	Linked	Iterator Based

Initialization	`Reset (collection);`
	`result :=`
	`Exp (CurrentItem(collection);`
	`collection := Get (collection);`
		`result :=`
		`Exp (collection^.item);`
		`p := collection^.next;`
			`iter:=collection.Iterator;`
			`result:= Exp (item);`

Loop Continuation Condition	`not EOF (collection)`	`p <> NIL`	`not iter.IsDone`

Loop Body	`if Exp(CurrentItem (collection)) >`[2]
	`result then`
	`result :=`
	`Exp(CurrentItem(collection))`
	`fi`
		`if Exp(p^item) >`[2]
		`result then`
		`result :=`
		`Exp(p^item))`
		`fi`
			`item := iter.Next;`
			`if Exp (item) >`[2]
			`result then`
			`result := Exp item)`
			`fi;`

Loop Update Statement	`colection :=` `Get (collection)`	`p := p^.next`	-
Post-mortem	The result is in `result`	The result is in `result`	The result is in `result`

Now you have an iterative solution for your problem.

10 Iterative Construction Quantification

(Also Known as: *Construction Loop Schema*)

You have used *Categorized Problem* and have categorized it as a *Map Quantification* problem. You have probably specified it with *Quantification Specification*. In your

[1] Implicit in Counted Loop in some programming languages.
[2] The comparison function depends on the type of the items in the collection and on the specific quantifier. '>' is for Max and '<' for Min.

problem, you have to compute the collection obtained after applying a function (named Exp) on every item in a collection that meets a property (named Fil). You are searching for an iterative solution to your problem.

The type of the items in the result collection are the same as the output type of the function Exp. The number of items in the result may be, at most, the same as the number of items in the input collection. In order to get the result, the collection has to be fully traversed. If the collection is empty, the result is a null collection. There is no precondition imposed on the collection of data by this quantifier. As the searched solution is iterative, you need to use a loop to compute the result.

Therefore, use the *Counted Loop* pattern [5] with the following design guideline for the loop parts outlined in it (we use as name for the traversal variable *i*, for the collection of data *collection* and for the variable with the result value *result*).

Loop Part	**Numeric Interval**	**Simple Linear**
Initialization	`i := a;` `Initialize result;`	`i := 0;` `Initialize result;`
Loop Continuation Condition	`i <= b` [1]	`i <= (Length of collection) - 1` [1]

Loop Body	`if Fil (i) then`
	`  Add Exp`
	`  (CurrentItem(collection))`
	`  to result`
	`fi`
		`if Fil (collection [i]) then`
		`  Add Exp (collection [i])) to result`
		`fi`

Loop Update Statement	`i := i + 1` [1]	`i := i + 1` [1]
Post-mortem	The result is in `result`	The result is in `result`

Loop Part	Streamed	**Linked**	**Iterator Based**

Initialization	`Reset (collection);`
	`Initialize result;`
		`p := collection;`
		`Initialize result`
			`iter:=`
			`  collection.Iterator;`
			`Initialize result`

Loop Continuation Condition	`not EOF (collection)`	`p <> NIL`	`not iter.IsDone`

Loop Body	`if Fil (CurrentItem (collection)) then`
	`  Add Exp(CurrentItem(collection))`
	`  to result`
	`fi`
		`if Fil (p^item) then`
		`  Add Exp(p^.item)`
		`  to result`
		`fi`
			`item := iter.Next;`
			`if Fil (item) then`
			`  Add Exp(item)`
			`  to result`
			`fi`

Loop Update Statement	`collection := Get (collection)`	`p := p^.next`	-
Post-mortem	The result is in `result`	The result is in `result`	The result is in `result`

The informal actions "`Initialize result`" and "`Add Exp()) to result`" are interpreted depending on the nature of the output collection (only *Simple Linear*, *Streamed* and *Linked* are considered) according to the guidelines on the following table. If the result collection is *Simple Linear* [5], use another local traversal variable (we use as name for this variable j) to index in the result collection. In such case, the length of the result is `j - 1`.

[1] Implicit in Counted Loop in some programming languages.

	Actions	
	Initialize result	*Add* `Exp( ) to result`
Simple Linear	`j := 0`	`Update (result, j, Exp (collection[i]); (1)` `j := j + 1;`
Streamed	`Rewrite (result)`	`Put (result,` `Exp (CurrentItem (collection)))`
Linked	`result := NIL`	`Insert (p^.item, result)` (2)

Now you have an iterative solution for your problem.

Acknowledgements

Thanks to Joseph Bergin, this paper´s EuroPLoP'2000 shepherd, for his valuable comments and suggestions. Also thanks to Jutta Eckstein, who shepherded an earlier version of this language at EuroPLoP'99.

References

[1] Bergin, J. Patterns for Selection. *Proceedings of the 4th European Conference on Patterns Languages of Programming and Computing*, 1999 (EuroPLoP´99). http://csis.pace.edu/~bergin/patterns/selection.html

[2] Clancy, M.J. and Linn, M.C. Patterns and Pedagogy. *Proceedings of the 30 th SIGCSE Technical Symposium on Computer Science Education*, 1999, 37-42.

[3] Gabriel, R. Simply Understood Code. http://c2.com/cgi/wiki?SimplyUnderstoodCode.

[4] Gries, D. The Science of Programming. *Springer-Verlag, Texts and Monographs in Computer Science*, 1981.

[5] Proulx, V. K. Programming Patterns and Design Patterns in the Introductory Computer Science Course, *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, 2000, 80-84. http://www.ccs.neu.edu/teaching/EdGroup/

[6] Soloway, E. From Problems to Programs Via Plans: The Content and Structure of Knowledge for Introductory Lisp Programming. *Journal of Educational Computing Research, 1(2)*, 1985, 157-172.

[7] Soloway, E. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29, 9 (Sept. 1986), 850-858.

[8] The Elementary Patterns Home Page: http://www.cs.uni.edu/%7Ewallingf/patterns/elementary/

[1] `Update (a, i, x)` updates `a` putting `x` in its ith element.
[2] Insertion in linked list.

Appendix A. Abstract Sequences

Concepts	Meaning	Concepts	Meaning
S : Seq (T)	Type declaration	*S (a . . b)*	< S(i) | i <? <a . . b> >
< >	Empty sequence	*IsEmpty (S)*	S = < >
<1, 3, 23, 0 >	Sequence literal	*First (S)*	Same as S[1]
< 'a' . . 'z' >	Sequence by interval	*Last (S)*	Same as S (Length (S))
< x^2 | x <? <1 . 10> >	Sequence intensionally defined	*Rest (S)*	Same as S (2 . . Length (S))
S [i]	I-th element of S	*Next (S, I)*	Same as S [i + 1]
Length (S)	Length of S	*Prev (S, I)*	Same as S (i – 1]
Domain (S)	<1 . . Length (S) >	*<? , =*	Predicates: membership, equality
Range (S)	< S(i) | i <? <1 .. Length (S)> >	*A ++ B*	Concatenate sequences A and B

Formal description and meanings for the *Abstract Sequence*