A Cognitive Model of Design Pattern Selection

Arturo Hinojosa

Submitted To:

Prof. Joshua Brett Tenenbaum
Department of Brain and Cognitive Sciences
Massachusetts Institute of Technology

Introduction: The Software Engineering Process

Design and implementation are the two main phases of software engineering. The decisions made during each phase are equally important to creating a viable system. Reckless execution of the engineering process can have a dramatic impact on the efficiency and quality of the final product. It is imperative for success that engineers consider all aspects of the challenge set before them and then craft a solution capable of handling any possible contingency. The solution must not only be able to expect the unexpected, but do so in a timely manner and in a fashion useful to the final user of the system. A brilliant design that takes an hour to compute a solution is of little use to a user who needs the information in minutes. Carefully fashioned and well defined designs are the foundation of good engineering, but even the most comprehensive and verbose design needs an equally appropriate implementation to be a successful solution. While a creative mind capable of abstract thought is the key to good and innovative designs, deciding on an implementation is a much more calculating process. The implementation must abide by the decisions made in the design, and must also balance the needs of the users with the operation of the system.

In spite of its design dependent nature, implementation can also be a very open ended problem with many possible solutions. Software engineers are usually only given a few diagrams and a very superficial outline of the system behavior. The system designer establishes the criteria for the operation of the system, but it is up to the programmer to select how to realize the design. The programmer must evaluate the desired system behavior and choose an implementation that not only meets the needs of the design, but does so efficiently given the requirements of the user and the established operational parameters. It is often the case that there is no single deterministic solution for the challenge. The same solution may have several different implementations, each with its own benefits and consequences. Some implementations are more appropriate under certain contexts, and knowing how to choose the best solution often takes years of experience and a heavy education in algorithm design, discrete mathematics, and general software engineering principles. The goal of this project is to develop a cognitive model of the process software engineers employ to select a specific implementation for a system design. Using inverse entailment and statistical data about real world implementation decisions, the aim of this research is to infer a set of general criteria that guide the engineer to the best possible implementation.

Background: The Gang of Four and Contemporary Research

Almost every software engineer is familiar with the design patterns established by the "Gang of Four," Gamma, Helm, Johnson, and Vlissides (1977). Their book is standard course material for software engineering classes and can be found on the office book shelves of countless programmers. They propose twenty three implementation patterns for common programming challenges that are widely accepted as the de-facto solutions when developing software. As Kung, Bhambhani, Shah, and Pancholi (2003) write in their paper, "design patterns provide a proven structure and characteristics for building highly maintainable and extensible software."

The Gang of Four divide their design patterns into three categories: Creational Patterns, Structural Patterns, and Behavioral Patterns. This project will focus on the final category, the behavioral patterns. This is the most diverse subgroup and has the most implementation

strategies. The design solutions in this category apply to the execution responsibilities of the software. It includes paradigms for data distribution, subsystem interfaces, and dynamic system behavior. Each pattern possesses a unique set of advantages that can be exploited under specific system conditions.

Because of the relative young nature of software engineering and design patterns, there has yet to be exhaustive research into the implications and criteria for design pattern selection. The most relevant effort has been by Kung, et al. (2003) to develop what they term an "Expert System for Suggesting Design Patterns," (ESSDP). Their approach is based on a user-query interface and a predefined knowledge base to systematically narrow selections based on the user provided information. They categorize their questions into five types based on significance and context. Each question carries a weight to help resolve conflicts. There approach can be outlined in five main steps:

1. Identify the circumstances in which a pattern can be applied.
2. Refine the circumstances with sub conditions.
3. Formulate questions to ask the user.
4. Classify the questions according to their level of significance.
5. Assign thresholds to patterns and weights to questions.

The model being developed in this project will attempt to adapt their approach in order to identify the features that are most revealing about pattern selection instead of trying to make a definite suggestion. Where as Kung, et al. (2003) studied the design patterns themselves, this research will focus on real world design choices and try to extrapolate some relationships based on those examples.

The cognitive model will employ the Progol programming language created by Muggleton (1995). Progol is an inductive logic programming language that uses inverse entailment to make assertions about classifications. It has been applied to research in machine learning, natural language processing, and other fields of artificial intelligence.

Theory: Mode Directed Inverse Entailment

The necessary approach to create any artificially intelligent system is to use the context provided by the subject world to infer new information about the causal relationships between the members of the world. Learning can happen by one of two methods. The first method is deduction, by which an observer uses a known set of true theories, T, to derive a set of consequences, E. Deductive inference is the result of repetitive applications of provable rules of inference. Each inference can be rationalized by using the previous assumptions. The second method is inductive learning, by which an observer uses the set of observed consequences to derive theories about their cause. Inductive inference may follow from the application of non provable and often incorrect rules of inference. The conclusions drawn from inductive inference have only the statistical support from a finite set of available data. The learner must sometimes need a probability distribution over the hypothesis space to assist them in their resolution. This type of Bayesian learning can help to solidify results and provides stronger statistical support.

Inductive logic programming implementations rely on inductive inference to develop a predicate basis for classification. Accurate results depend on the amount of noise in the data set and having a statistically significant diverse sampling of the information. Background

knowledge guides the classification process along with the examples. Each statement from the background knowledge is treated as a conjecture. These conjectures are the foundation for a new set of conjectures which try to explain the examples. According to Muggleton (1995) there are five main approaches to creating an inductive logic programming system:

1. Inverse resolution in propositional logic.
2. Inverse resolution in first order definite clause logic.
3. Determinate relative least general generalization.
4. Inverse implication.
5. Mode directed inverse entailment.

Each of these is based on Duce's six rules for inductive inference. The most relevant for this research are the final two rules, intra-construction (D.1) and inter-construction (D.2).

D.1

p←A,B		p←A,C
q←B	p←A,q	q←C

D.2

p←A,B		p←A,C
p←r,B	r←A	q←r,C

These rules apply single depth inversion to resolve relationships between the variables and constants. It is possible then to create a hierarchal mechanism for deriving the example set from background knowledge. This is the basic foundation for the Progol learning engine. The underlying theory of Progol is explained by Muggleton (1995) in his introductory paper. Inductive logic systems use a background knowledge base, B, and a set of examples, E, to derive a hypothesis, H, about the relationships between the data. This can be stated formally as:

$$\textbf{I.1}\ B \wedge H \models E$$

The goal of the learning is to derive the simplest hypothesis H which is consistent across all of the examples. Each clause of the hypothesis space must explain at least one of the examples. If this were not true, then the clause would be redundant and H would not be the simplest explanation. Furthermore, by constraining H and E to be single Horn clauses, and applying the law of contraposition, $\neg H$ and $\neg E$ are found to be ground skolemized unit clauses. With the existential quantifiers removed, and all that remains are universal assertions about the examples and hypothesis.

$$\textbf{I.2}\ B \wedge \neg E \models \neg H$$

If $\neg C$ is a conjunction of ground literals true for all models of $B \wedge \neg E$, and since $\neg H$ must be true for all models by definition, it possible to further conclude that:

$$\textbf{I.3}\ B \wedge \neg E \models \neg C \models \neg H$$

$$\textbf{I.4}\ C \models H$$

This relationship between the conjunctive clause and the hypothesis implies that a subset solution for H can be derived by taking those clauses that Θ-subsume C. A complete solution for all candidates in H can therefore be calculated by considering all the clauses that Θ-subsume the sub-saturants of C.

The derivation rests on model theory as opposed to resolution proof theory, meaning learning is achieved through observation in contrast to deduction. Because of the allowable error tolerated in this type of learning, this more general approach is optimal in the case of loose categorization because the system does not try to over burden itself trying to resolve contradictions in the data. Given the subjective nature of design pattern selection, this type of flexibility is important.

Design: Creating a Cognitive Model

The foundation for this research is to use positive examples from real world situations to draw inferences about the specific characteristics of the data that lead to categorization. While Gold demonstrated in his 1967 paper that positive examples where insufficient to fully capture and learn a language (think of language in this case as an abstract set of rules, or a grammar, which can define a design pattern selection), Muggleton (1997) shows that positive examples can be enough to guide learning to an acceptable level of low error. Gold's argument is based on the fact that for every set of positive examples, there are at least two candidate hypotheses. The first is the language of all possible descriptions, and the second is the more complex language which exactly accepts the examples. The approach taken by Progol is to use these two languages as bounds for a search and come to a compromise which can categorize the data using the simplest hypothesis. To do so, the model must be able to perform the following:

1. Use a set of features that can fully define the examples and allow the engine to discriminate between their characteristics.
2. Develop a relationship mapping between the features of the examples and the design patterns they employ.
3. Make qualitative assessments of the relevance of each feature and make rational decisions about its role in categorization.
4. Express each example as a compilation of simple predicate statements.

The most important step in designing the model is assessing the characteristic features of the different examples. These features must reflect the demands on the software system, the static and dynamic behavior of the system, as well as the function of the system. The examples come from different sources, and inherently carry the bias of the original designer. Where one designer chose to use one implementation, another designer might well have chosen another. It is important then to choose features that are specific enough to reveal the characteristics of the problem, yet are general enough so the learning focuses on why the designer chose to use that implementation, as opposed to why another designer would have chosen another.

Many external factors, not accounted for in this model, have significant effects on the designer's choice. Experience with certain patterns, preferences for certain patterns, and other environmental influences all factor into the design choice. This model attempts to take an empirical view of implementation process, and tries to minimize the influence of these effects by studying several examples for each design pattern and using multiple sources.

Materials and Apparatus: Implementing a Progol System

The goal of the prolog model is to determine a set of parameters which decide a design pattern implementation given a set of requirements on the system. A prolog system is composed of four distinct sections:

1. Mode declarations (both head and body)
2. Type declarations
3. Background knowledge
4. Examples

Mode Declarations

This section defines the predicate relationships for the logic system. These declarations explicitly state the form and necessary arguments for each type of predicate statement. Using these declarations, the Progol system establishes the syntax rules for the rest of the model. Mode declarations have two types: head and body. Head declarations are the predicate relation the system is trying to decide. Body declarations are supporting predicates that are relevant to the head declaration. These declarations take the form of:

:- mode_type(recall, predicate_name(#/+/-argument_1, #/+/-argument_2,…, #/+/-argument_n).

The recall argument is a number or symbol that declares a bound for the possible simultaneous solutions to the predicate statement. If the bound is unknown, the symbol '*' can be used to declare an unbound predicate statement. The symbols that precede each argument declare if it is an input variable, output variable, or constant. Alternatively, instead of the place-marker variable, the mode declaration can also use a normal statement as an argument input. A normal statement is bracketed tuple of terms, either constants or variables, that can be taken as input. The mode declarations state not only what type of predicates can be explicitly declared by the user, but also the type of predicates that can be introduced by the engine during the learning process.

Type Declarations

This section assigns subject types for the logic system. It is an exhaustive declaration of all the categories of objects in the model over the set of entities. While each type does not have to be defined in any sense, all model subjects must be assigned a type. The declarations have the form of:

type(subject)

The type declarations are unary predicate statements that assert that a subject satisfies the requirements to be considered of a certain type.

Procedure: The Progol Learning Engine and Analysis of Output

The progol learning engine is an iterative mechanism that searches out the most specific hypothetical clause to define the relationships declared as head modes. It begins by checking that all the assertions made in the model file are consistent, and no contradictions exist. Then, a subsumption lattice is created from the background knowledge and examples declared in the file. The creation of each conjecture in the subsumption lattice is as follows:

1. The engine examines the first positive horn clause example found in the file, *e*, such that a :- $b_1, b_2, \ldots, b_n$.
2. It negates the clause *e* to create a skolemized conjunctive normal form expression, $\neg e$, such that $\neg a \wedge b_1 \wedge b_2 \wedge \ldots \wedge b_n$.
3. This new expression is added to the background knowledge cache.
4. The engine then finds the first head mode declaration *h* that Θ-subsumes *a*.
5. For each substitution in Θ:
    a. If the substitution is over a constant, then the skolem variable is substituted in *h*.
    b. If the substitution is over a variable, then a hash table entry for the skolem variable is substituted into *h*. Additionally, if the substitution is over an input variable, then the skolem variable is cached in the set *IT*.
6. Finally the head mode declaration *h* is added to the overall conjecture *C*.
7. Next, the engine examines all the body mode declarations.
8. For each body mode declaration *b*:
    a. The engine attempts a substitution over all the input variables with the cached skolem variables in *IT*.
    b. For the number of possible solutions (defined by the recall number):
    c. If the set of substitution satisfies the predicate:
        i. if the substitution is over a constant, then the skolem variable is substituted in the clause.
        ii. If the substitution is over a variable, then a hash table entry for the skolem variable is substituted into the clause. Additionally, if the variable is an output variable, then the skolem variable is added to the cache *IT*.
    d. Finally, the negated body mode declaration is added to the overall conjecture *C*.
9. This process is repeated until the maximum variable depth has been reached.

The final output of these iterations is a disjunctive clause with *h* as its head and the negated *b* expressions as its body atoms. For each example then, trying to find a suitable hypothesis involves a search of the bounded sub-lattice, constrained by the empty hypothesis and the conjecture *C*. The progol engine builds the hypothesis by trying decreasingly general clauses until a comprehensive one is found. This search can be seen as a descent through a tree whose root node is the empty conjecture and whose trunk nodes are refinements. Progol uses an A* style search to find the node with the maximum compression. The search uses a heuristic measurement to guide the learning calculated from the following data:

1. p_s = The number of positive examples correctly deduced by the candidate clause.
2. n_s = The number of negative examples incorrectly deduced by the candidate clause.
3. c_s = The length of candidate clause – 1

4. h_s = The number of additional atoms necessary to complete the cause, as calculated by inspecting the output and counting the number atoms needed to connect any skolem variables.
5. f_s = The heuristic measure given by $p_s - n_s - c_s - h_s$.

The search algorithm works as follows:

1. Maintain two caches, *open*, initially populated with the empty clause, and the empty cache *closed*.
2. *s* is set to the member of *open* that maximizes f_s,
3. *s* is removed from *open*.
4. *s* is added to *closed*.
5. Check the value of n_s. If it is 0 and $f_s > 0$, no possible improvements to the heuristic measure are possible therefore skip to step 8.
6. Otherwise, descend down the tree.
7. Add the next refinement nodes to *open* and remove any members common to *closed*.
8. If *s* maximizes f_s for all clauses in *closed*, $n_s = 0$, and $f_s(s) > f_s$ for all members of open, then we have found the optimal solution and no further searching is required.
9. If the open cache is empty, then no generalization was possible and algorithm returns the original clause, otherwise the algorithm returns to step 2.

This algorithm is guaranteed to terminate and return the most comprehensive clause with minimal length. The program also has an internal mechanism for dealing with redundant examples and background knowledge. This mechanism is not covered in this paper as it does not pertain to the learning, but is available from the Progol documentation.

Work Plan: Implementing a Cognitive Model for Design Pattern Selection.

The first step in designing the model is implementing the Progol system. The outline below details the specific steps necessary to implement the cognitive model.

1. Implement Progol learning.
    a. Gather examples of associations between system behavior and an optimal design pattern implementation.
    b. Examine the examples and select a set of relevant and revealing features of the system behavior.
        i. Include features common to only a few examples.
        ii. Include features common to many examples.
        iii. Translate each feature into a predicate expression.
    c. Create the background knowledge by declaring predicate statements about all the system behaviors and the various features.
    d. Run the progol learning engine to generate a most specific clause for each design pattern.
2. Analyze the most specific clauses for each design pattern.

The following are the head and body mode declarations used in the Progol program to learn about the design pattern features. They were chosen based on personal experience and the works of Muggleton (1995), and Budinsky, Finnie, Vlissides, and Yu (1996). The code can be found in its entirety in Appendix A. In the interest of avoiding repetition, a comprehensive explanation of each feature and the different examples can be found in the comments of the code.

```
:- modeh(1,optimal_pattern(+prob_stat, #design_pattern))?
:- modeb(1,distributed(+prob_stat))?
:- modeb(1,daemon(+prob_stat))?
:- modeb(1,tracks_state(+prob_stat))?
:- modeb(1,tracks_value(+prob_stat))?
:- modeb(1,tracks_ext_state(+prob_stat))?
:- modeb(1,dep_order(+prob_stat))?
:- modeb(1,dep_input(+prob_stat))?
:- modeb(1,dep_impl(+prob_stat))?
:- modeb(1,dep_hist(+prob_stat))?
:- modeb(1,for_interp(+prob_stat))?
:- modeb(1,for_extension(+prob_stat))?
:- modeb(1,for_input(+prob_stat))?
:- modeb(1,for_output(+prob_stat))?
:- modeb(1,for_prolif(+prob_stat))?
:- modeb(1,for_monit(+prob_stat))?
:- modeb(1,behavior( +prob_stat, #rtbehavior ))?
```

Results: The Conclusions Derived from the Model

Using the examples from the data collected from other's research and all available reference materials, the following are the conclusions of the Progol system about the relevant features for each design pattern:

```
optimal_pattern(A,chain_of_resp) :- distributed(A), dep_order(A),
        dep_input(A), for_output(A), behavior(A,dynamic).

optimal_pattern(A,command) :- dep_input(A), for_extension(A),
        for_input(A), for_output(A), for_prolif(A), for_monit(A),
        behavior(A,static).

optimal_pattern(A,interpreter) :- dep_order(A), dep_input(A),
        for_input(A), behavior(A,dynamic).

optimal_pattern(A,iterator) :- tracks_ext_state(A), dep_order(A),
        dep_input(A), dep_impl(A), dep_hist(A), for_input(A),
        for_monit(A), behavior(A,dynamic).

optimal_pattern(A,mediator) :- distributed(A), for_input(A),
        behavior(A,static).

optimal_pattern(A,memento) :- distributed(A), daemon(A), tracks_state(A),
        dep_order(A), dep_hist(A), for_monit(A), behavior(A,
        static).

optimal_pattern(A,observer) :- distributed(A), tracks_value(A),
        dep_input(A), for_output(A), for_monit(A), behavior(A,
        static).
```

```
optimal_pattern(A,state) :- behavior(A,dynamic).

optimal_pattern(A,strategy) :- behavior(A,static).

optimal_pattern(A,templete) :- behavior(A,dynamic).

optimal_pattern(A,visitor) :- behavior(A,static).
```

These results closely reflect the type of assertions I anticipated before running the model. I used several different data sets, and tried different feature combinations before accepting the results above. Using the built in Progol analysis mechanism, I generated a set of contingency tables that state the anticipated accuracy of the model. While all these tables state a 100% overall accuracy, I believe that is due to an insufficient data set and over classification.

Discussion and Conclusion: What the Model Means and Relevance to Future Research

This model represents a good start in understanding the causal relationships between implementation selection and features of the solutions they endeavor to undertake. Other research I found took many different approaches to addressing design pattern selections. Some researchers are beginning to study the UML representations of the different design patterns to gain insight into the implications of the internal mechanics of each pattern, while others continue to work on direct classification.

What made this project interesting was the unique approach taken to learn about the design pattern features. As opposed to most contemporary research which takes a deductive approach to classification, the Progol engine allowed me to make inductive inferences about the classification. This type of learning seems especially helpful in this case given my own biases about each design pattern and when it should be used. By letting the data talk for itself, I learned a lot about machine learning, as well as some more about software engineering in the process.

References

Budinsky, F., Finnie, M., Vlissides, J., & Yu, P. (1996). Automatic code generation from design patterns, IBM Systems Journal, Vol. 35, No. 2.

Gamme, E., Helm R., Johnson R., & Vlissides, J. (1977). Design Patterns: Elements of Reusable Object Orientated Software. Boston, MA: Addison-Wesley.

Kung, D., Bhambhani, H., Shah, R., & Pancholi, G. (2003). An Expert System for Suggesting Design Patterns – A Methodology and a Prototype. In Software Engineering With Computational Intelligence. Kluwer International Series in Engineering and Computer Science.

Muggleton, S. (1995). Inverse Entailment and Progol, New Generation Computing Journal, Vol. 13, pp. 245-286.

Muggleton, S. (1997). Learning from positive data, Proceedings of the Sixth International Workshop on Inductive Logic Programming, Springer-Verlag, LNAI 1314.

Appendix A: Progol Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Input file for cognative modeling of design pattern selection
% Arturo Hinojosa
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- set(r,100).
:- set(h,1000).

%%%%%%%%%%%%%%%%%%%%%%%%
% Mode delcarations
%%%%%%%%%%%%%%%%%%%%%%%%

% Head Mode
%%%%%%%%%%%%%%%%%%%%%%%%

%%
% States that a single output should be given when queried for
% the optimal design pattern choice.
%
% Input: prob_state A problem statement definition.
% Output: The optimal design pattern.
%%
:- modeh(1,optimal_pattern(+prob_stat, #design_pattern))?


% Body Modes
%%%%%%%%%%%%%%%%%%%%%%%%

:- modeb(1,distributed(+prob_stat))?
:- modeb(1,daemon(+prob_stat))?
:- modeb(1,tracks_state(+prob_stat))?
:- modeb(1,tracks_value(+prob_stat))?
:- modeb(1,tracks_ext_state(+prob_stat))?
:- modeb(1,dep_order(+prob_stat))?
:- modeb(1,dep_input(+prob_stat))?
:- modeb(1,dep_impl(+prob_stat))?
:- modeb(1,dep_hist(+prob_stat))?
:- modeb(1,for_interp(+prob_stat))?
:- modeb(1,for_extension(+prob_stat))?
:- modeb(1,for_input(+prob_stat))?
:- modeb(1,for_output(+prob_stat))?
:- modeb(1,for_prolif(+prob_stat))?
:- modeb(1,for_monit(+prob_stat))?
:- modeb(1,behavior( +prob_stat, #rtbehavior ))?

%%%%%%%%%%%%%%%%%%%%%%%%
% Type declarations
%%%%%%%%%%%%%%%%%%%%%%%%

%
% Problem statments
%
% These are various software tasks to be analyzed.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%
% Provide context sensative help for UI screen.
%
% 1. User clicks somewhere on screen.
% 2. Program iterates through location context to find
%    the right help output.
% 3. Searches inorder of generality.
%%
prob_stat(context_help).

%%
% An event that has multiple consequences.
%
% 1. Given some random event:
% 2. Program must iterate through series of subsystems
%    notifying them of the event.
%%
prob_stat(event_handler).

%%
% System for approving business expenses. Based on the
% quantity and type of the expense, a hierarchiel progression
% is made to find a certified supervisor who can approve the
% expense.
%
%%
prob_stat(exp_appr).

%%
% Store DVD lib info.
%
%%
prob_stat(dvd_lib).

%%
% Spam Filter.
%
%%
prob_stat(spam_filt).

%%
% Perform action triggered by external event.
%
% 1. Given some randomly occuring external event:
% 2. Triggers a system responce.
%%
prob_stat(action_listener).

%%
% Model a transaction.
%
% 1. Collect information about transaction.
% 2. Create a transaction to another process or subsystem.
%%
prob_stat(model_trans).
```

```
%%
% Caclulator capable of undoing transactions.
%
%%
prob_stat(trak_calc).

%%
% Parse a literal text string.
%
% 1. Given some string input:
% 2. Parse the string into subsections.
% 3. Recursively parse subsections.
% 4. Decide the most basic subsection meanings.
%%
prob_stat(parse_text).

%%
% Evaluate a formula.
%
% 1. Given some formula
% 2. Parse the formula into hierarchy of operations.
% 3. Evaluate each subproblem
% 4. Output the formula value.
%
%%
prob_stat(eval_formula).

%%
% Convert a roman numeral to a decimal.
%
%%
prob_stat(conv_rom).

%%
% Examine the contents of a set.
%
% 1. Given any random set:
% 2. Inspect all elements of the set individually.
% 3. The set is responsible for providing a traversing scheme.
%
%%
prob_stat(exam_set).

%%
% Count the members of a set.
%
% 1. Given any random set:
% 2. Count the number of elements in the set.
% 3. The set is responsible for providing a traversing scheme.
%
%%
prob_stat(count_set).

%%
% Cycle through a set skipping certain elements.
%
%%
```

```
prob_stat(disc_iter).

%%
% Collect user input from a composite dialog box.
%
% 1. Given control over some composite dialog box:
% 2. Collect the user input from each field.
% 3. Cache the user input.
% 4. Modify fields based on other field values.
%
%%
prob_stat(dialog_input).

%%
% Manage a system resource.
%
% 1. Monitor a system resource.
% 2. When system changed, propogate change to all associated resources.
% 3. Maintain system consistency.
%
%%
prob_stat(man_resc).

%%
% A chat room coordinator
%
%%
prob_stat(chat_host).

%%
% Implement an undo feature.
%
% 1. Allow the user to undo an operation.
%
%%
prob_stat(make_undo).

%%
% Implement history in a web browser.
%
% 1. Allow a user to scroll through the history of viewed pages.
%
%%
prob_stat(brow_hist).

%%
% Use a single interface to display multiple
% potential client information.
%
%%
prob_stat(sale_disp).

%%
% Display content linked to a menu
%
% 1. User has a list associated with individual appearance schemes.
% 2. Each time the user changes their selection, change scheme.
```

```
%
%%
prob_stat(disp_cont).

%%
% Display information for a highlighted file in a file manager.
%
% 1. User has several selection options.
% 2. System monitors current selection and displays information about the
selection.
%
%%
prob_stat(disp_info).

%%
% Allow client to chose one of many sorting algorithms.
%
%%
prob_stat(many_sort).

%%
% Maintain a remote connection port with various protocols.
%
% 1. System maintains an external communication resource.
% 2. Connections take many forms, and system must handle all types.
%
%%
prob_stat(comm_hand).

%%
% Create a drawing/editing tool with multiple discrete functions.
%
% 1. User can select different drawing tools.
% 2. Behavior of tool varies with selection.
%
%%
prob_stat(draw_tool).

%%
% Line break controller for text editor.
%
% 1. Text editor uses system to insert line breaks.
% 2. System must count content width.
% 3. Content can be text, pictures, graphs, etc.
%
%%
prob_stat(line_break).

%%
% Interest calculator.
%
% 1. A calculator for interest for different programs.
% 2. Programs have different interest rate and policy.
%
%%
prob_stat(intr_calc).
```

```
%%
% XML parser.
%
% 1. Given some XML content tree:
% 2. Inspect tree nodes.
% 3. Cache content and inspect children nodes.
%
%%
prob_stat(xml_parse).

%%
% Graphics renderer
%
% 1. Given some graphic input:
% 2. Distribute input to different graphics engines.
%
%%
prob_stat(render).

%%
% Compute total costs from multiple sources.
%
% 1. Queury different subsystems for costs.
%.2. Output sum of all subsystem costs.
% 3. Each subsystem has a different interface.
%
%%
prob_stat(comp_cost).

%%
% Add a new discount feature to cost calculator.
%
% 1. Using an existing calculator system.
% 2. After calculation, apply a variable discount.
% 3. Discount is context specific.
%
%%
prob_stat(add_disc).

%%
% Alerts registered investors of stock price change.
%
%%
prob_stat(tell_inv).

%%
% Modify bank account based on standing and funds.
%
%%
prob_stat(bank_acct).

%%
% A single line of communication for differenct connection
% types and rates.
%
%%
prob_stat(comm_line).
```

```
%%
% HR resource to update employee information.
%
%%
prob_stat(hr_update).


%%
% Searching DVD rating archieve.
%
%%
prob_stat(dvd_arch).

%%
% Resolves string matches for archieve query.
%
%%
prob_stat(res_str).

%%
% Browse DVD collection.
%
%%
prob_stat(dvd_brow).

%%
% Pass communication from DVD db to UI.
%
%%
prob_stat(dvd_intr).

%%
% Creates a compressed back up of the DVD information.
%
%%
prob_stat(dvd_back).

%%
% Updates system with new release information.
%
%%
prob_stat(dvd_rel).

%%
% DVD quality rankings.
%
%%
prob_stat(dvd_rank).

%%
% Variable DVD info display.
%
%%
prob_stat(dvd_disp).

%%
```

```
% Uniform info display for different media.
%
%%
prob_stat(med_disp).

%%
% Display a bunch of DVD blurbs.
%
%%
prob_stat(blurbs).

%%
% Security system manager.
%
%%
prob_stat(secr_man).

%%
% Message handler.
%
%%
prob_stat(msg_hand).

%%
% Inventory tracking.
%
%%
prob_stat(inv_track).

%%
% Exception handling.
%
%%
prob_stat(exp_hand).

%%
% Graphics validation.
%
%%
prob_stat(pict_val).

%%
% Get project info.
%
%%
prob_stat(get_info).

%%
% Ticket master...find cheapest open seat.
%
%%
prob_stat(find_open).

%%
% Electronic light switch signals.
%
%%
```

```
prob_stat(light).

%%
% Window manager message.
%
%%
prob_stat(win_msg).

%%
% Db entry.
%
%%
prob_stat(db_entry).

%%
% Login result.
%
%%
prob_stat(login).

%%
% XML editor.
%
%%
prob_stat(xml_edit).

%%
% Syntax highligher for IDE.
%
%%
prob_stat(ide_hili).

%%
% Command processor
%
%%
prob_stat(com_proc).

%%
% MP3 decoder.
%
%%
prob_stat(mp3_dec).

%%
% Regex matching.
%
%%
prob_stat(regex_match).

%%
% Give a uniform interface without any implementation details.
%
%%
prob_stat(hide_impl).

%%
```

```
% A stack.
%
%%
prob_stat(stack).

%%
% A video service for video confrencing.
%
%%
prob_stat(video).

%%
% A postfix converter.
%
%%
prob_stat(postfix).

%%
% A binary search tree.
%
%%
prob_stat(bst).

%%
% A dynamic tree manager.
%
%%
prob_stat(dyn_tree).

%%
% A safe IO system manager.
%
%%
prob_stat(safe_io).

%%
% A thread manager.
%
%%
prob_stat(threads).

%%
% Save a file.
%
%%
prob_stat(save_file).

%%
% Account history.
%
%%
prob_stat(acct_hist).

%%
% Modular arithmatic.
%
%%
```

```
prob_stat(mod_math).

%%
% Alarm system.
%
%%
prob_stat(alarm).

%%
% IO driver.
%
%%
prob_stat(io_driver).

%%
% Message router.
%
%%
prob_stat(msg_rout).

%%
% CMOS logic circuit,
%
%%
prob_stat(cmos).

%%
% HR db entry
%
%%
prob_stat(hr_entry).

%%
% Finite state machine
%
%%
prob_stat(fsm).

%%
% Traffic light
%
%%
prob_stat(tr_lite).

%%
% Toll booth.
%
%%
prob_stat(toll).

%%
% A CMOS logic gate.
%
%%
prob_stat(cmos_gate).

%%
```

```
% An auto belt diagnostic program.
%
%%
prob_stat(auto_belt).

%%
% A monopoly game piece.
%
%%
prob_stat(mono_chip).

%%
% A monopoly game piece.
%
%%
prob_stat(bed_test).

%%
% Prioritizer
%
%%
prob_stat(prior).

%%
% Logrithmic math.
%
%%
prob_stat(log_math).

%%
% Dynamic interpreter.
%
%%
prob_stat(dyn_interp).

%%
% Bucket painting.
%
%%
prob_stat(buck_pnt).

%%
% Rearrange data collection.
%
%%
prob_stat(rearg).

%%
% Dispatch information.
%
%%
prob_stat(dispatch).

%%
% Format a string ( -> ALL CAPS ).
%
%%
```

```
prob_stat(form_str).



% Design Patterns
%%%%%%%%%%%%%%%%%%%%%

design_pattern(chain_of_resp).
design_pattern(command).
design_pattern(interpreter).
design_pattern(iterator).
design_pattern(mediator).
design_pattern(memento).
design_pattern(observer).
design_pattern(state).
design_pattern(strategy).
design_pattern(template).
design_pattern(visitor).


%%
% The requirements on the run-time system behavior:
%
%     static: System behavior is determinstic during run-time.
%     dynamic: System behavior depends on run-time choices.
%
%%
rtbehavior( static ).
rtbehavior( dynamic ).

%%%%%%%%%%%%%%%%%%%%%%
% Positive Examples
%%%%%%%%%%%%%%%%%%%%%%

optimal_pattern( context_help, chain_of_resp).
optimal_pattern( event_handler, chain_of_resp).
optimal_pattern( exp_appr, chain_of_resp).
optimal_pattern( dvd_lib, chain_of_resp).
optimal_pattern( spam_filt, chain_of_resp).
optimal_pattern( secr_man, chain_of_resp).
optimal_pattern( msg_hand, chain_of_resp).
optimal_pattern( inv_track, chain_of_resp).
optimal_pattern( exp_hand, chain_of_resp).
optimal_pattern( pict_val, chain_of_resp).
optimal_pattern( get_info, chain_of_resp).
optimal_pattern( find_open, chain_of_resp).

optimal_pattern( action_listener, command ).
optimal_pattern( model_trans, command ).
optimal_pattern( trak_calc, command ).
optimal_pattern( dvd_arch, command ).
optimal_pattern( light, command ).
optimal_pattern( win_msg, command ).
optimal_pattern( db_entry, command ).
optimal_pattern( login, command ).
optimal_pattern( xml_edit, command ).
```

```
optimal_pattern( ide_hili, command ).
optimal_pattern( com_proc, command ).

optimal_pattern( parse_text, interpreter ).
optimal_pattern( eval_formula, interpreter ).
optimal_pattern( conv_rom, interpreter ).
optimal_pattern( res_str, interpreter ).
optimal_pattern( mp3_dec, interpreter ).
optimal_pattern( regex_match, interpreter ).
optimal_pattern( hide_impl, interpreter ).
optimal_pattern( postfix, interpreter ).

optimal_pattern( exam_set, iterator ).
optimal_pattern( count_set, iterator ).
optimal_pattern( disc_iter, iterator ).
optimal_pattern( dvd_brow, iterator ).
optimal_pattern( stack, iterator ).
optimal_pattern( bst, iterator ).

optimal_pattern( dialog_input, mediator ).
optimal_pattern( man_resc, mediator ).
optimal_pattern( chat_host, mediator ).
optimal_pattern( dvd_intr, mediator ).
optimal_pattern( dyn_tree, mediator ).
optimal_pattern( safe_io, mediator ).
optimal_pattern( threads, mediator ).

optimal_pattern( brow_hist, memento ).
optimal_pattern( make_undo, memento ).
optimal_pattern( sale_disp, memento ).
optimal_pattern( dvd_back, memento ).
optimal_pattern( video, memento ).
optimal_pattern( save_file, memento ).
optimal_pattern( acct_hist, memento ).

optimal_pattern( disp_cont, observer ).
optimal_pattern( disp_info, observer ).
optimal_pattern( tell_inv, observer ).
optimal_pattern( dvd_rel, observer ).
optimal_pattern( mod_math, observer ).
optimal_pattern( alarm, observer ).
optimal_pattern( io_driver, observer ).
optimal_pattern( msg_rout, observer ).
optimal_pattern( cmos, observer ).

optimal_pattern( comm_hand, state ).
optimal_pattern( draw_tool, state ).
optimal_pattern( bank_acct, state ).
optimal_pattern( dvd_rank, state ).
optimal_pattern( hr_entry, state ).
optimal_pattern( fsm, state ).

optimal_pattern( line_break, strategy ).
optimal_pattern( intr_calc, strategy ).
optimal_pattern( many_sort, strategy ).
optimal_pattern( dvd_disp, strategy ).
optimal_pattern( bed_test, strategy ).
```

```
optimal_pattern( xml_parse, templete ).
optimal_pattern( render, templete ).
optimal_pattern( comm_liner, templete ).
optimal_pattern( med_disp, templete ).
optimal_pattern( prior, templete ).


optimal_pattern( comp_cost, visitor ).
optimal_pattern( add_disc, visitor ).
optimal_pattern( hr_update, visitor ).
optimal_pattern( blurbs, visitor ).
optimal_pattern( buck_pnt, visitor ).
optimal_pattern( dispatch, visitor ).
optimal_pattern( form_str, visitor ).


%%%%%%%%%%%%%%%%%%%%%%
% Background Knowledge
%%%%%%%%%%%%%%%%%%%%%%


%%
% Subsystem is triggered by random mouse click.
% Subsystem must choose one of multiple help subsystems.
% Each help subsystem passes on request to next most general
% help subsystem.
% Least general subsystem choosen.
% Subsystem displays help message.
%%
distributed( context_help ).
dep_input( context_help ).
dep_order( context_help ).
behavior( context_help, dynamic ).
for_output( context_help ).

%%
% Randomly occuring event has series of consequences.
%
%%
distributed( event_handler ).
dep_order( event_handler ).
behavior( event_handler, static ).
for_prolif( event_handler ).

%%
% Iterative check of approving business expense.
%
%%
dep_order( exp_appr ).
dep_input( exp_appr ).
behavior( exp_appr, dynamic ).
for_output( exp_appr ).

%%
% Add DVD to inventory.
%
```

```
%%
distributed( dvd_lib ).
dep_order( dvd_lib ).
for_input( dvd_lib ).
for_prolif( dvd_lib ).
behavior( dvd_lib, static ).

%%
% Spam Filter.
%
%%
dep_order( spam_filt ).
dep_input( spam_filt ).
for_interp( spam_filt ).
for_extension( spam_filt ).
for_input( spam_filt ).
behavior( spam_filt, dynamic ).

%%
% Must associate a static action with an input device.
% Action executed when input triggered.
%
%%
dep_input( action_listener ).
for_extension( action_listener ).
for_input( action_listener ).
for_output( action_listener ).
for_prolif( action_listener ).
for_monit( action_listener ).
behavior( action_listener, static ).

%%
% Organize various values into a set.
%
%%
dep_input( model_trans ).
for_extension( model_trans ).
for_input( model_trans ).
for_output( model_trans ).
for_prolif( model_trans ).
behavior( model_trans, dynamic ).

%%
% Calculator that can track history.
%
%%
tracks_state( trak_calc ).
dep_order( trak_calc ).
dep_input( trak_calc ).
dep_hist( trak_calc ).
for_output( trak_calc ).
behavior( trak_calc, dynamic ).

%%
% Recursively parse a literal text string.
% Breakdown into substrings.
% Start with most general substring types.
```

```
% Use specific evaulation and parser for each substring type.
%
%%
dep_order( parse_text ).
dep_input( parse_text ).
for_inter( parse_text ).
for_input( parse_text ).
behavior( parse_text, dynamic ).

%%
% Parse formula into operations tree.
% Evaluate each operation.
% Start with base operations and work way up.
% Use specific evaluation for each operation type.
% Output value of formula.
%
%%
tracks_value( eval_formula ).
tracks_ext_state( eval_formula ).
dep_order( eval_formula ).
dep_input( eval_formula ).
for_interp( eval_formula ).
for_input( eval_formula ).
for_output( eval_formula ).
behavior( eval_formula, dynamic ).

%%
% Converts a roman numeral to a decimal.
%
%%
tracks_value( conv_rom ).
tracks_ext_state( conv_rom ).
dep_order( conv_rom ).
dep_input( conv_rom ).
for_interp( conv_rom ).
for_input( conv_rom ).
for_output( conv_rom ).
behavior( conv_rom, dynamic ).


%%
% Examine the contents of a random set.
% Each set implementation must be traversable via the same method.
%
%%
tracks_ext_state( exam_set ).
dep_order( exam_set ).
dep_input( exam_set ).
dep_impl( exam_set ).
dep_hist( exam_set ).
for_input( exam_set ).
for_monit( exam_set ).
behavior( exam_set, dynamic ).

%%
% Count the elements of a random set.
% Each set implementation must be countable via the same method.
```

```
%
%%
tracks_value( count_set ).
tracks_ext_state( count_set ).
dep_order( count_set ).
dep_input( count_set ).
dep_impl( count_set ).
dep_hist( count_set ).
for_output( count_set ).
behavior( count_set, dynamic ).

%%
% Iterates through a set, skipping certain
% elements.
%
%%
tracks_value( disc_iter ).
tracks_ext_state( disc_iter ).
dep_order( disc_iter ).
dep_input( disc_iter ).
dep_impl( disc_iter ).
dep_hist( disc_iter ).
for_input( disc_iter ).
for_monit( disc_iter ).
behavior( disc_iter, dynamic ).

%%
% 1. Control elements of composite dialog.
% 2. Dynamic UI behavior.
% 3. User randomly clicks action button.
% 4. Collects field values from all elements.
%
%%
distributed( dialog_input ).
for_input( dialog_input ).
behavior( dialog_input, static ).

%%
% 1. Monitor resources.
% 2. Changes must be communicated between resources.
% 3. Resource values must be consistent.
%
%%
distributed( man_resc ).
daemon( man_resc ).
tracks_state( man_resc ).
tracks_value( man_resc ).
dep_hist( man_resc ).
for_prolif( man_resc ).
for_monit( man_resc ).
behavior( man_resc, dynamic ).


%%
% A chat room host, coordinates conversations.
%
%%
```

```
distributed( chat_host ).
daemon( chat_host ).
for_input( chat_host ).
for_output( chat_host ).
for_prolif( chat_host ).
for_monit( chat_host ).
behavior( chat_host, static ).

%%
% 1. Maintain a record of the activity history.
% 2. Browse history.
% 3. Change display for history movement.
%
%%
distributed( brow_hist ).
daemon( brow_hist ).
tracks_state( brow_hist ).
dep_order( brow_hist ).
dep_hist( brow_hist ).
for_monit( brow_hist ).
behavior( brow_hist, static ).

%%
% 1. Maintain a record of the state history.
% 2. Browse history.
% 3. Change state for history movement.
%
%%
distributed( make_undo ).
daemon( make_undo ).
tracks_state( make_undo ).
dep_order( make_undo ).
dep_hist( make_undo ).
for_monit( make_undo ).
behavior( make_undo, static ).

%%
% Sales display.
%
%%
distributed( sale_disp ).
tracks_state( sale_disp ).
dep_input( sale_disp ).
dep_hist( sale_disp ).
for_output( sale_disp ).
behavior( sale_disp, dynamic ).


%%
% 1. Display content associated with a list member.
% 2. List member selected randomly by user.
%
%%
distributed( disp_cont ).
tracks_value( disp_cont ).
dep_input( disp_cont ).
for_output( disp_cont ).
```

```
for_monit( disp_cont ).
behavior( disp_cont, static ).

%%
% 1. Display file information for user.
% 2. User randomly selects a file.
%
%%
distributed( disp_info ).
tracks_value( disp_info ).
dep_input( disp_info ).
for_output( disp_info ).
for_monit( disp_info ).
behavior( disp_info, static ).


%%
% 1. Control communication subsystems.
% 2. Monitor connection channel.
% 3. Support multiple connection protocols.
% 4. Vary subsystem behavior to fit comm type.
% 5. Dispatch information to appropriate subsystems.
%%
distributed( comm_hand ).
daemon( comm_hand ).
tracks_state( comm_hand ).
tracks_ext_state( comm_hand ).
dep_input( comm_hand ).
dep_impl( comm_hand ).
for_input( comm_hand ).
for_output( comm_hand ).
for_prolif( comm_hand ).
for_monit( comm_hand ).
behavior( comm_hand, dynamic ).

%%
% 1. Graphics tool that can vary behavior.
% 2. Different user toolbox selections modify behavior.
% 3. Each behavior has unique engine.
%.4. User randomly changes behavior.
%
%%
distributed( draw_tool ).
daemon( draw_tool ).
tracks_state( draw_tool ).
dep_input( draw_tool ).
for_output( draw_tool ).
behavior( draw_tool, dynamic ).

%%
% 1. Monitor line contents for threshold value.
% 2. Randomly occuring event triggers line break.
%
%%
daemon( line_break ).
tracks_state( line_break ).
tracks_value( line_break ).
```

```
dep_input( line_break ).
dep_hist( line_break ).
for_monit( line_break ).
behavior( line_break, dynamic ).


%%
% Let client choose one of many sorting algorithms.
%
%%
tracks_state( many_sort ).
tracks_value( many_sort ).
dep_order( many_sort ).
dep_input( many_sort ).
for_output( many_sort ).
behavior( many_sort, dynamic ).

%%
% 1. Calculates interest in various ways.
% 2. Outputs interest.
%
%%
tracks_state( intr_calc ).
tracks_value( intr_calc ).
dep_input( intr_calc ).
for_output( intr_calc ).
behavior( intr_calc, dynamic ).

%%
% 1. Parses XML content tree.
% 2. Iterates down parent and children.
% 3. Examines node and caches content.
% 4. Different node types use different operation.
%
%%
tracks_state( xml_parse ).
tracks_ext_state( xml_parse ).
dep_order( xml_parse ).
dep_input( xml_parse ).
for_interp( xml_parse ).
for_input( xml_parse ).
for_output( xml_parse ).
behavior( xml_parse, dynamic ).


%%
% 1. Send relevant information to different graphics engines.
% 2. Certain graphics types only use some engines.
%
%%
tracks_state( render ).
tracks_ext_state( render ).
dep_order( render ).
dep_input( render ).
for_monit( render ).
for_prolif( render ).
behavior( render, dynamic ).
```

```
%%
% 1. Collects cost values from various independent subsystems.
% 2. Each subsystem has different interface.
%
%%
distributed( comp_cost ).
tracks_value( comp_cost ).
dep_order( comp_cost ).
dep_input( comp_cost ).
for_output( comp_cost ).
behavior( comp_cost, dynamic ).

%%
% 1. Add a dynamic discount feature to an existing program.
%
%%
distributed( comp_cost ).
dep_input( comp_cost ).
for_output( comp_cost ).
behavior( comp_cost, dynamic ).

%%
% Tell investors of a change in stock price.
%
%%
distributed( tell_inv ).
daemon( tell_inv ).
tracks_value( tell_inv ).
dep_input( tell_inv ).
for_output( tell_inv ).
for_monit( tell_inv ).
for_prolif( tell_inv ).
behavior( tell_inv, static ).


%%
% Dynamic bank account.
%
%%
daemon( bank_acct ).
tracks_state( bank_acct ).
tracks_value( bank_acct ).
dep_input( bank_acct ).
for_prolif( bank_acct ).
for_monit( bank_acct ).
behavior( bank_acct, dynamic ).


%%
% Dynamic communication line.
%
%%
distributed( comm_line ).
daemon( comm_line ).
tracks_state( comm_line ).
tracks_value( comm_line ).
```

```
dep_input( comm_line ).
for_input( comm_line ).
for_output( comm_line ).
for_prolif( comm_line ).
for_monit( comm_line ).
behavior( comm_line, dynamic ).


%%
% Update HR db info.
%
%%
distributed( hr_update ).
for_output( hr_update ).
behavior( hr_update, dynamic ).


%%
% Access DVD archieve.
%
%%
dep_input( dvd_arch ).
for_input( dvd_arch ).
for_output( dvd_arch ).
for_prolif( dvd_arch ).
behavior( dvd_arch, dynamic ).



%%
% Resolve search string.
%
%%
dep_order( res_str ).
dep_input( res_str ).
for_interp( res_str ).
for_input( res_str ).
for_prolif( res_str ).
behavior( res_str, dynamic ).


%%
% Browse DVD collection.
%
%%
tracks_value( dvd_brow ).
tracks_ext_state( dvd_brow ).
dep_order( dvd_brow ).
dep_input( dvd_brow ).
for_output( dvd_brow ).
behavior( dvd_brow, dynamic ).

%%
% DVD interface,
%
%%
dep_input( dvd_intr ).
```

```
for_input( dvd_intr ).
for_output( dvd_intr ).
for_prolif( dvd_intr ).
for_monit( dvd_intr ).
behavior( dvd_intr, static ).


%%
% Creates compressed back up
%
%%
tracks_ext_state( dvd_back ).
dep_input( dvd_back ).
for_input( dvd_back ).
behavior( dvd_back, static ).


%%
% Updates system with new release info.
%
%%
distributed( dvd_rel ).
tracks_value( dvd_rel ).
dep_input( dvd_rel ).
for_output( dvd_rel ).
for_prolif( dvd_rel ).
for_monit( dvd_rel ).
behavior( dvd_rel, static ).


%%
% Ranking system for DVDs
%
%%
tracks_state( dvd_rank ).
tracks_value( dvd_rank ).
for_output( dvd_rank ).
behavior( dvd_rank, static ).

%%
% Variable DVD info display.
%
%%
distributed( dvd_disp ).
dep_input( dvd_disp ).
for_output( dvd_disp ).
behavior( dvd_disp, dynamic ).

%%
% Uniform media info display.
%
%%
distributed( med_disp ).
dep_input( med_disp ).
for_output( med_disp ).
behavior( med_disp, dynamic ).
```

```
%%
% Display many info blurbs.
%
%%
distributed( blurbs ).
for_output( blurbs ).
behavior( blurbs, dynamic ).

%%
% Security system manager.
%
%%
distributed( secr_man ).
daemon( secr_man ).
dep_order( secr_man ).
dep_input( secr_man ).
for_input( secr_man ).
for_monit( secr_man ).
for_prolif( secr_man ).
behavior( secr_man, dynamic ).


%%
% MFC Message handler.
%
%%
distributed( msg_hand ).
daemon( msg_hand ).
dep_input( msg_hand ).
for_input( msg_hand ).
for_output( msg_hand ).
for_prolif( msg_hand ).
behavior( msg_hand, dynamic ).

%%
% Inventory tracker.
%
%%
distributed( inv_track ).
tracks_ext_state( inv_track ).
dep_order( inv_track ).
for_output( inv_track ).
for_monitoring( inv_track ).
behavior( inv_track, static ).


%%
% Exception handling.
%
%%
distributed( exp_hand ).
daemon( exp_hand ).
tracks_state( exp_hand ).
dep_order( exp_hand ).
dep_input( exp_hand ).
dep_hist( exp_hand ).
```

```
for_output( exp_hand ).
for_prolif( exp_hand ).
for_monit( exp_hand ).
behavior( exp_hand, dynamic ).


%%
% Graphics validation.
%
%%
distributed( pict_val ).
daemon( pict_val ).
dep_order( pict_val ).
for_prolif( pict_val ).
for_monit( pict_val ).
behavior( pict_val, static ).


%%
% Get info from project hierarchy.
%
%%
distributed( get_info ).
dep_order( get_info ).
dep_input( get_info ).
for_output( get_info ).
behavior( get_info, static ).


%%
% Finds cheapest open seat for venue.
%
%%
dep_order( find_open ).
for_output( find_open ).
behavior( find_open, static ).

%%
% Light switch operation,
%
%%
dep_input( light ).
for_input( light ).
for_output( light ).
for_prolif( light ).
behavior( light, dynamic ).


%%
% Window manager message.
%
%%
dep_input( win_msg ).
for_input( win_msg ).
for_output( win_msg ).
for_prolif( win_msg ).
behavior( win_msg, dynamic ).
```

```
%%
% Database entry.
%
%%
dep_input( db_entry ).
for_input( db_entry ).
for_output( db_entry ).
for_prolif( db_entry ).
behavior( db_entry, dynamic ).

%%
% Login result.
%
%%
dep_input( login ).
for_input( login ).
behavior( login, dynamic ).


%%
% XML editor.
%
%%
tracks_state( xml_edit ).
tracks_ext_state( xml_edit ).
for_interp( xml_edit ).
for_extension( xml_edit ).
for_input( xml_edit ).
behavior( xml_edit, dynamic ).


%%
% IDE text high lighter.
%
%%
daemon( ide_hili ).
tracks_values( ide_hili ).
dep_input( ide_hili ).
for_interp( ide_hili ).
for_extension( ide_hili ).
behavior( ide_hili, dynamic ).


%%
% Command processor.
%
%%
dep_input( com_proc ).
for_interp( com_proc ).
for_extension( com_proc ).
for_input( com_proc ).
behavior( com_proc, dynamic ).
```

```
%%
% MP3 Decoder
%
%%
dep_order( mp3_dec ).
dep_input( mp3_dec ).
for_interp( mp3_dec ).
for_input( mp3_dec ).
behavior( mp3_dec, dynamic ).


%%
% Regular expression matching.
%
%%
tracks_value( regex_match ).
dep_order( regex_match ).
dep_input( regex_match ).
for_interp( regex_match ).
for_input( regex_match ).
behavior( regex_match, dynamic ).

%%
% Hide implementation from clients.
%
%%
tracks_value( hide_impl ).
tracks_state( hide_impl ).
tracks_ext_state( hide_impl ).
dep_impl( hide_impl ).
for_extension( hide_impl ).
for_input( hide_impl ).
for_output( hide_impl ).
for_prolif( hide_impl ).
behavior( hide_impl, dynamic ).


%%
% A stack.
%
%%
tracks_state( stack ).
dep_order( stack ).
dep_hist( stack ).
dep_impl( stack ).
for_output( stack ).
behavior( stack, dynamic ).

%%
% A video confrencer with memory.
%
%%
distributed( video ).
tracks_ext_state( video ).
dep_hist( video ).
for_input( video ).
for_output( video ).
```

```
for_monit( video ).
behavior( video, static ).


%%
% A postfix converter.
%
%%
dep_order( postfix ).
dep_input( postfix ).
for_interp( postfix ).
for_input( postfix ).
for_prolif( postfix ).
behavior( postfix, dynamic ).


%%
% A binary search tree.
%
%%
tracks_state( bst ).
dep_order( bst ).
dep_input( bst ).
dep_hist( bst ).
for_input( bst ).
for_output( bst ).
for_monit( bst ).
behavior( bst, dynamic ).



%%
% A dynamic tree manager.
%
%%
distributed( dyn_tree ).
tracks_state( dyn_tree ).
dep_input( dyn_tree ).
for_monit( dyn_tree ).
behavior( dyn_tree, dynamic ).


%%
% A safe IO manager.
%
%%
distributed( safe_io ).
daemon( safe_io ).
tracks_state( safe_io ).
dep_input( safe_io ).
dep_hist( safe_io ).
for_prolif( safe_io ).
for_monit( safe_io ).
behavior( safe_io, dynamic ).


%%
```

```
% A thread manager.
%
%%
distributed( threads ).
daemon( threads ).
tracks_state( threads ).
dep_order( threads ).
dep_hist( threads ).
for_prolif( threads ).
for_monit( threads ).
behavior( threads, dynamic ).

%%
% Save a file.
%
%%
tracks_ext_state( save_file ).
dep_hist( save_file ).
for_input( save_file ).
behavior( save_file, static ).


%%
% Account History
%
%%
tracks_ext_state( acct_hist ).
dep_order( acct_hist ).
dep_hist( acct_hist ).
for_input( acct_hist ).
for_monit( acct_hist ).
behavior( acct_hist, static ).

%%
% Modular Math
%
%%
tracks_value( disp_cont ).
dep_input( disp_cont ).
for_output( disp_cont ).
behavior( disp_cont, dynamic ).


%%
% Alarm system
%
%%
distributed( alarm ).
daemon( alarm ).
tracks_state( alarm ).
dep_input( alarm ).
for_output( alarm ).
for_monit( alarm ).
for_prolif( alarm ).
behavior( alarm, dynamic ).
```

```
%%
% IO Monitor
%
%%
distributed( io_driver ).
daemon( io_driver ).
tracks_state( io_driver ).
tracks_ext_state( io_driver ).
dep_input( io_driver ).
for_input( io_driver ).
for_output( io_driver ).
for_monit( io_driver ).
for_prolif( io_driver ).
behavior( io_driver, dynamic ).

%%
% Message router.
%
%%
distributed( msg_rout ).
daemon( msg_rout ).
dep_input( msg_rout ).
for_input( msg_rout ).
for_output( msg_rout ).
for_monit( msg_rout ).
for_prolif( msg_rout ).
behavior( msg_rout, dynamic ).


%%
% CMOS logic ciruict.
%
%%
distributed( cmos ).
daemon( cmos ).
tracks_state( cmos ).
tracks_value( cmos ).
dep_order( cmos ).
dep_input( cmos ).
dep_hist( cmos ).
for_input( cmos ).
for_output( cmos ).
for_monit( cmos ).
for_prolif( cmos ).
behavior( cmos, dynamic ).


%%
% HR db entry.
%
%%
tracks_state( hr_entry ).
dep_input( hr_entry ).
for_input( hr_entry ).
for_output( hr_entry ).
for_prolif( hr_entry ).
```

```
for_monit( hr_entry ).
behavior( hr_entry, dynamic ).


%%
% A FSM
%
%%
distributed( fsm ).
tracks_state( fsm ).
dep_input( fsm ).
for_input( fsm ).
for_monit( fsm ).
behavior( fsm, static ).


%%
% Bed tester for a couple.
%
%%
tracks_state( bed_test ).
tracks_value( bed_test ).
dep_input( bed_test ).
for_output( bed_test ).
behavior( bed_test, dynamic ).


%%
% Prioritizer
%
%%
tracks_ext_state( prior ).
dep_order( prior ).
dep_input( prior ).
for_output( prior ).
behavior( prior, dynamic ).



%%
% Bucket painting.
%
%%
distributed( buck_pnt ).
for_output( buck_pnt ).
behavior( buck_pnt, static ).


%%
% Dispatch
%
%%
distributed( dispatch ).
daemon( dispatch ).
dep_input( dispatch ).
for_prolif( dispatch ).
behavior( dispatch, static ).
```

```
%%
% Format string.
%
%%
dep_input( form_str ).
for_prolif( form_str ).
behavior( form_str, static ).
```