

Chapter 10

Artificial Intelligence: Cryptanalysis

Sentient creatures exhibit a vastly complex set of behaviors that spring from the mind through mechanisms that we only poorly understand. For example, think about how you solve the problem of planning a route through a city to run a set of errands. Consider also how, when walking through a dimly lit room, you are able to recognize the boundaries of objects and avoid stumbling. Furthermore, think about how you can focus on one conversation at a party while dozens of people are talking simulta- neously. None of these kinds of problems lends itself to a straightforward algorithmic solution. Optimal route planning is known to be an NP- complete problem. Navigating through dark terrain involves deriving understanding from visual input that is (very literally) fuzzy and incom- plete. Identifying a single speaker from dozens of sources requires that the listener distinguish meaningful data from noise and then filter out all unwanted conversations from the remaining cacophony.

Researchers in the field of artificial intelligence have pursued these and similar problems to improve our understanding of human cognitive pro- cesses. Activity in this field often involves the construction of intelligent systems that mimic certain aspects of human behavior. Erman, Lark, and Hayes-Roth point out that:

intelligent systems differ from conventional systems by a number of attributes, not all of which are always present:

* They pursue goals which vary over time.
* They incorporate, use, and maintain knowledge.
* They exploit diverse, ad hoc subsystems embodying a variety of selected methods.

**413**

* They interact intelligently with users and other systems.
* They allocate their own resources and attention. [1]

Any one of these properties is sufficiently demanding to make the crafting of intelligent systems a very difficult task. When we consider that intelli- gent systems are being developed for a variety of domains that affect both life and property, such as for medical diagnosis or aircraft routing, the task becomes even more demanding because we must design these systems so that they are never actively dangerous: Artificial intelligences rarely embody any kind of commonsense knowledge.

Although the field has at times been oversold by an overly enthusiastic press, the study of artificial intelligence has given us some very sound and practical ideas, among which we count approaches to knowledge repre- sentation and the evolution of common problem-solving architectures for intelligent systems, including rule-based expert systems and the black- board model [2]. In this chapter, we turn to the design of an intelligent sys- tem that solves cryptograms using a blackboard framework in a manner that parallels the way a human would solve the same problem. As we will see, the use of object-oriented development is very well suited to this domain.

# Inception

Our problem is one of cryptanalysis, the process of transforming ciphertext back to plaintext. In its most general form, deciphering cryptograms is an intractable problem that defies even the most sophisticated techniques. Happily, our problem is relatively simple because we limit ourselves to single substitution ciphers.

## Cryptanalysis Requirements

Cryptography “embraces methods for rendering data unintelligible to unautho- rized parties” [3]. Using cryptographic algorithms, messages (plaintext) may be transformed into cryptograms (ciphertext) and back again.

One of the most basic kinds of cryptographic algorithms, employed since the time of the Romans, is called a substitution cipher. With this cipher, every letter of the plaintext alphabet is mapped to a different letter. For example, we might shift every letter to its successor: A becomes B, B becomes C, Z wraps around to become A, and so on. Thus, the plaintext

CLOS is an object-oriented programming language

may be enciphered to the cryptogram

DMPT jt bo pckfdu-psjfoufe qsphsbnnjoh mbohvbhf

Most often, the substitution of letters is jumbled. For example, A becomes G, B

becomes J, and so on. As an example, consider the following cryptogram:

PDG TBCER CQ TCK AL S NGELCH QZBBR SBAJG

Hint: The letter C represents the plaintext letter O.

It is a vastly simplifying assumption to know that only a substitution cipher was employed to encode a plaintext message; nevertheless, deciphering the resulting cryptogram is not an algorithmically trivial task. Deciphering sometimes requires trial and error, wherein we make assumptions about a particular substitution and then evaluate their implications. For example, we may start with the one- and two-letter words in the cryptogram and hypothesize that they stand for common words such as I and a, or it, in, is, of, or, and on. By substituting the other occurrences of these ciphered letters, we may find hints for deciphering other words. For instance, if there is a three-letter word that starts with o, the word might reasonably be one, our, or off.

We can also use our knowledge of spelling and grammar to attack a substitution cipher. For example, an occurrence of double letters is not likely to represent the sequence qq. Similarly, we might try to expand a word ending with the letter g to the suffix ing. At a higher level of abstraction, we might assume that the sequence of words it is is more likely to occur than the sequence if is. Also, we might assume that the structure of a sentence typically includes a noun and a verb. Thus, if our analysis has identified a verb but no actor or agent, we might start a search for adjectives and nouns.

Sometimes we may have to backtrack. For example, we might have assumed that a certain two-letter word was or, but if the substitution for the letter r causes contradictions or blind alleys in other words, we might have to try the word of or on instead and consequently undo other assumptions we had based on this earlier substitution.

This leads us to the overarching requirement of our problem: to devise a system that, given a cryptogram, transforms it back to its original plaintext, assuming that only a simple substitution cipher was employed.

## Defining the Boundaries of the Problem

As part of our analysis, let’s walk through a scenario of solving a simple crypto- gram. Spend the next few minutes solving the following problem, and as you pro- ceed, record how you did it (no fair reading ahead!):

Q AZWS DSSC KAS DXZNN DASNN

As a hint, we note that the letter W represents the plaintext V.

Trying an exhaustive search is pretty much senseless. Assuming that the plaintext alphabet encompasses only the 26 uppercase English characters, there are 26! (approximately 4.03  1026) possible combinations. Thus, we must try something other than a brute force attack. An alternate technique is to make an assumption based on our knowledge of sentence, word, and letter structure and then follow this assumption to its natural conclusions. Once we can go no further, we choose the next most promising assumption that builds on the first one, and so on, as long as each succeeding assumption brings us closer to a solution. If we find that we are stuck, or we reach a conclusion that contradicts a previous one, we must back- track and alter an earlier assumption.

Here is our solution, showing the results at each step.

1. According to the hint, we may directly substitute V for W.

Q AZVS DSSC KAS DXZNN DASNN

1. The first word is small, so it is probably either an A or an I; let’s assume that it is an A.

A AZVS DSSC KAS DXZNN DASNN

1. The third word needs a vowel, and it is likely to be the double letters. It is probably neither II nor UU, and it can’t be AA because we have already used an A. Thus, we might try EE.

A AZVE DEEC KAE DXZNN DAENN

1. The fourth word is three letters long and ends in an E; it is likely to be the word THE.

A HZVE DEEC THE DXZNN DHENN

1. The second word needs a vowel, but only an I, O, or U (we’ve already used

A and E). Only the I gives us a meaningful word.

A HIVE DEEC THE DXINN DHENN

1. There are few four-letter words that have a double E, including DEER, BEER, and SEEN. Our knowledge of grammar suggests that the third word should be a verb, and so we select SEEN.

A HIVE SEEN THE SXINN SHENN

1. This sentence is not making any sense (hives cannot see), so we probably made a bad assumption somewhere along the way. The problem seems to lie with the vowel in the second word, so we might consider reversing our ini- tial assumption.

I HAVE SEEN THE SXANN SHENN

1. Let’s attack the last word. The double letters can’t be SS (we’ve used an S, and besides, SHESS doesn’t make any sense), but LL forms a meaningful word.

I HAVE SEEN THE SXALL SHELL

1. The fifth word is part of a noun phrase and so is probably an adjective (STALL, for example, is rejected on this account). Searching for words that fit the pattern S?ALL yields SMALL.

I HAVE SEEN THE SMALL SHELL

Thus, we have reached a solution.

We may make the following observations about this problem-solving process.

* + - We applied many different sources of knowledge, such as knowledge about grammar, spelling, and vowels.
    - We recorded our assumptions in one central place and applied our sources

of knowledge to these assumptions to reason about their consequences.

* + - We reasoned opportunistically. At times, we reasoned from general to spe- cific rules (if the word is three letters long and ends in E, it is probably THE), and at other times, we reasoned from the specific to the general (?EE? might be DEER, BEER, or SEEN, but since the word must be a verb and not a noun, only SEEN satisfies our hypothesis).

From these problem-solving observations, we can identify some key abstractions. Key abstractions are analysis elements of our solution that begin to establish the initial architectural framework. The three bullets identify multiple knowledge sources, a central place for assumptions or hypotheses, and a control component that opportunistically controls the problem solving.

What we have described is a problem-solving approach known as a *blackboard model*. The blackboard model was first proposed by Newell in 1962 and later incorporated by Reddy and Erman into the Hearsay and Hearsay II projects, both of which dealt with the problems of speech recognition [4]. The blackboard model proved to be useful in this domain, and the framework was soon applied successfully to other domains, including signal interpretation, the modeling of three-dimensional molecular structures, image understanding, and planning [5]. Blackboard frameworks have proven to be particularly noteworthy with regard to the representation of declarative knowledge and are space and time efficient when compared with alternate approaches [6].

The blackboard framework is an architectural pattern that can be applied as a result of the analysis of our problem-solving algorithm. The framework can be represented in terms of classes and mechanisms that describe how instances of those classes collaborate.

## The Architecture of the Blackboard Framework

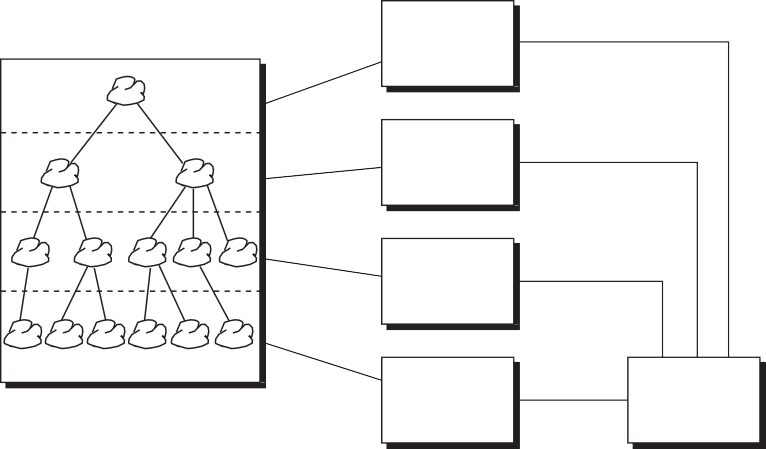
Englemore and Morgan explain the blackboard model by analogy to the problem of a group of people solving a jigsaw puzzle:

Imagine a room with a large blackboard and around it a group of people each holding over-size jigsaw pieces. We start with volunteers who put on the black- board (assume it’s sticky) their most “promising” pieces. Each member of the group looks at his pieces and sees if any of them fit into the pieces already on the blackboard. Those with the appropriate pieces go up to the blackboard and update the evolving solution. The new updates cause other pieces to fall into place, and other people go to the blackboard to add their pieces. It does not mat- ter whether one person holds more pieces than another. The whole puzzle can be solved in complete silence; that is, there need be no direct communication among the group. Each person is self-activating, knowing when his pieces will contrib- ute to the solution. No a priori established order exists for people to go up to the blackboard. The apparent cooperative behavior is mediated by the state of the solution on the blackboard. If one watches the task being performed, the solution is built incrementally (one piece at a time) and opportunistically (as an opportu- nity for adding a piece arises), as opposed to starting, say, systematically from the left top corner and trying each piece. [7]

As Figure 10–1 indicates, the blackboard framework consists of three elements: a blackboard, multiple knowledge sources, and a controller that mediates among these knowledge sources [8]. Notice how the following description describes the key abstractions identified from the problem space. According to Nii, “the pur- pose of the blackboard is to hold computational and solution-state data needed by and produced by the knowledge sources. The blackboard consists of objects from the solution space. The objects on the blackboard are hierarchically organized into levels of analysis. The objects and their properties define the vocabulary of the solution space” [9].

As Englemore and Morgan explain, “The domain knowledge needed to solve a problem is partitioned into knowledge sources that are kept separate and indepen- dent. The objective of each knowledge source is to contribute information that will lead to a solution to the problem. A knowledge source takes a set of current information on the blackboard and updates it as encoded in its specialized knowl- edge. The knowledge sources are represented as procedures, sets of rules, or logic assertions” [10].

Knowledge Sources



Blackboard

KS

KS

KS

KS

Control

**Figure 10–1** A Blackboard Framework

Knowledge sources, or KSs for short, are domain-specific. In speech recognition systems, knowledge sources might include agents that can reason about pho- nemes, morphemes, words, and sentences. In image recognition systems, knowl- edge sources would include agents that know about simple picture elements, such as edges and regions of similar texture, as well as higher-level abstractions repre- senting the objects of interest in each scene, such as houses, roads, fields, cars, and people. Generally speaking, knowledge sources parallel the hierarchical structure of objects on the blackboard. Furthermore, each knowledge source uses objects at one level as its input and then generates and/or modifies objects at another level as its output. For instance, in a speech recognition system, a knowl- edge source that embodies knowledge about words might look at a stream of phonemes (at a low level of abstraction) to form a new word (at a higher level of abstraction). Alternately, a knowledge source that embodies knowledge about sentence structure might hypothesize the need for a verb (at a high level of abstraction); by filtering a list of possible words (at a lower level of abstraction), this knowledge source can verify the hypothesis.

These two approaches to reasoning represent forward-chaining and backward- chaining, respectively. Forward-chaining involves reasoning from specific asser- tions to a general assertion, and backward-chaining starts with a hypothesis, then tries to verify the hypothesis from existing assertions. This is why we say that control in the blackboard model is opportunistic: Depending on the circum- stances, a knowledge source might be selected for activation that uses either for- ward- or backward-chaining.

Knowledge sources usually embody two elements, namely, preconditions and actions. The preconditions of a knowledge source represent the state of the blackboard in which the knowledge source shows an interest. For example, a precondition for a

knowledge source in an image recognition system might be the discovery of a relatively linear region of picture elements (perhaps representing a road). Trigger- ing a precondition causes the knowledge source to focus its attention on this part of the blackboard and then take action by processing its rules or procedural knowledge.

Under these circumstances, polling is unnecessary: When a knowledge source thinks it has something interesting to contribute, it notifies the blackboard con- troller. Figuratively speaking, it is as if each knowledge source raises its hand to indicate that it has something useful to do; then, from among eager knowledge sources, the controller calls on the one that looks the most promising.

## Analysis of Knowledge Sources

Let’s return to our specific problem and consider the knowledge sources that can contribute to a solution. As is typical with most knowledge-engineering applica- tions, the best strategy is to sit down with an expert in the domain and record the heuristics that this person applies to solve the problems in the domain. For our present problem, this might involve trying to solve a number of cryptograms and recording our thinking process along the way.

Our analysis suggests that 13 knowledge sources are relevant; they appear with the knowledge they embody in the following list:

* + - Common prefixes Common word beginnings such as re, anti, and un
    - Common suffixes Common word endings such as ly, ing, es, and ed
    - Consonants Nonvowel letters
    - Direct substitution Hints given as part of the problem statement
    - Double letters Common double letters, such as tt, ll, and ss
    - Letter frequency Probability of the appearance of each letter
    - Legal strings Legal and illegal combinations of letters, such as qu

and zg, respectively

* + - Pattern matching Words that match a specified pattern of letters
    - Sentence structure Grammar, including the meanings of noun and verb

phrases

* + - Small words Possible matches for one-, two-, three-, and four-

letter words

* + - Solved Whether or not the problem is solved, or if no further progress can be made
    - Vowels Nonconsonant letters
    - Word structure The location of vowels and the common structure of

nouns, verbs, adjectives, adverbs, articles, conjunc- tives, and so on

From an object-oriented perspective, each of these 13 knowledge sources repre- sents a candidate class in our architecture: Each instance embodies some state (its knowledge), each exhibits certain class-specific behavior (a suffix knowledge source can react to words suspected of having a common ending), and each is uniquely identifiable (a small-word knowledge source exists independent of the pattern-matching knowledge source).

We may also arrange these knowledge sources in a hierarchy. Specifically, some knowledge sources operate on sentences, others on letters, still others on contigu- ous groups of letters, and the lowest-level ones on individual letters. Indeed, this hierarchy reflects the objects that may appear on the blackboard: sentences, words, strings of letters, and letters.

# Elaboration

We are now ready to design a solution to the cryptanalysis problem using the blackboard framework we have described. This is a classic example of reuse-in- the-large, in that we are able to reuse a proven architectural pattern as the founda- tion of our design.

The architecture of the blackboard framework suggests that among the highest- level objects in our system are a blackboard, several knowledge sources, and a controller. Our next task is to identify the domain-specific classes and objects that specialize these general key abstractions.

## Blackboard Objects

The blackboard is an elaborate structure of multiple levels of abstractions. The abstractions are captured as objects that appear hierarchically on a blackboard structure. The hierarchical object structure parallels the different levels of abstrac- tions of the knowledge sources. The knowledge sources use the blackboard as a global source of input data, partial solutions, alternatives, final solutions, and con- trol information.

To begin the design of the blackboard’s hierarchical structure, we identify the fol- lowing classes:

* + - Sentence A complete cryptogram
    - Word A single word in the cryptogram
    - CipherLetter A single letter of a word

Knowledge sources must also share knowledge about the assumptions each

makes, so we include the following class of blackboard objects:

* + - Assumption An assumption made by a knowledge source

Finally, it is important to know what plaintext and ciphertext letters in the alpha- bet have been used in assumptions made by the knowledge sources, so we include the following class:

* + - Alphabet The plaintext alphabet, the ciphertext alphabet, and the

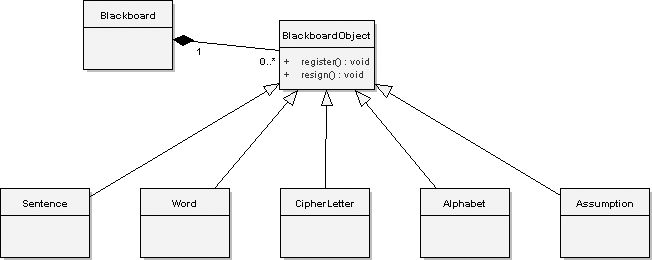
mapping between the two

Is there anything in common among these five classes? We answer with a resounding yes: Each one of these classes represents objects that may be placed on a blackboard, and that very property distinguishes them from, for example, knowledge sources and controllers. Thus, we invent the BlackboardObject class as the superclass of every object that may appear on a blackboard. Figure 10–2 shows our preliminary design of the Blackboard abstraction.

Looking at the BlackboardObject class from its outside view, we may define two applicable operations:

* + - register Add the object to the blackboard.
    - resign Remove the object from the blackboard.

Why do we define register and resign as operations on instances of BlackboardObject, instead of on the Blackboard itself? This situation is not unlike telling an object to draw itself in a window. The litmus test for deciding where to place these kinds of operations is whether or not the class itself has sufficient knowledge or responsibility to carry out the operation. In the case of



**Figure 10–2** The Preliminary Blackboard Class Diagram Design

register and resign, this is indeed the case: The BlackboardObject is the only abstraction with detailed knowledge of how to attach or remove itself from the Blackboard (although it certainly does require collaboration with the BlackboardObject). In fact, it is an important responsibility of this abstrac- tion that each BlackboardObject be self-aware as it is attached to the Blackboard because only then can it begin to participate in opportunistically solving the problem on the Blackboard.

## Dependencies and Affirmations

Individual sentences, words, and cipher letters have another thing in common: Each has certain knowledge sources that depend on it. A given knowledge source may express an interest in one or more of these objects, and therefore, a sentence, word, or cipher letter must maintain a reference to each such knowledge source, so that when an assumption about the object changes, the appropriate knowledge sources can be notified that something interesting has happened. To provide this mechanism, we introduce a simple abstract class: Dependent.

To design the Dependent class, we include an object that represents a collec- tion of knowledge sources:

* + - references Collection of knowledge sources

In addition, the following operations are defined for this class:

* + - add Add a reference to the knowledge source.
    - remove Remove a reference to the knowledge source.
    - numberOfDependents Return the number of dependents.
    - notify Broadcast an operation of each dependent.

The operation notify has the semantics of a passive iterator, meaning that when we invoke it, we can supply an operation that we wish to perform on every depen- dent in the collection.

Dependency is an independent property that can be mixed in with other classes. For example, a CipherLetter is a BlackboardObject as well as a Dependent, so we can combine these two abstractions to achieve the desired behavior. Using an abstract class in this way increases the reusability and separa- tion of concerns in our architecture.

CipherLetter and Alphabet have another property in common: Instances of both of these classes may have assumptions made about them (and remember that an Assumption object is also a kind of BlackboardObject). For

example, a certain knowledge source might assume that the ciphertext letter K represents the plaintext letter P. As we get closer to solving our problem, we might make the unchangeable assertion that G represents J. Thus we need to include a class that maintains the assumptions and assertions about the associated object. This class we will identify as Affirmation.

In our architecture, we will make affirmations only about individual letters, as in CipherLetter and Alphabet. As our earlier scenario implied, cipher letters represent single letters about which statements might be made, and alphabets comprise many letters, each of which might have different statements made about them. Defining Affirmation as an independent class thus captures the com- mon behavior across these two disparate classes.

We define the following operations for instances of the Affirmation class:

* + - make Make a statement.
    - retract Retract a statement.
    - ciphertext Given a plaintext letter, return its ciphertext equivalent.
    - plaintext Given a ciphertext letter, return its plaintext equivalent.

Further analysis suggests that we should clearly distinguish between the two roles played by a statement: An assumption, which represents a temporary mapping between a ciphertext letter and its plaintext equivalent, and an assertion, which is a permanent mapping, meaning that the mapping is defined and therefore not changeable. During the solution of a cryptogram, knowledge sources will make many assumptions, and as we move closer to a final solution, these mappings eventually become assertions. To model these changing roles, we will refine the previously identified class Assumption and introduce a new subclass named Assertion, both of whose instances are managed by instances of the class Affirmation as well as placed on the blackboard. We begin by completing the signature of the operations make and retract to include an Assumption or Assertion argument, and then add the following selectors:

* + - isPlainLetterAsserted A selector: Is the plaintext letter

defined?

* + - isCipherLetterAsserted A selector: Is the ciphertext letter

defined?

* + - plainLetterHasAssumption A selector: Is there an assumption

about the plaintext letter?

* + - cipherLetterHasAssumption A selector: Is there an assumption

about the ciphertext letter?

Assumption objects are a kind of BlackboardObject because they repre- sent state that is of general interest to all knowledge sources. Member objects will need to be declared to represent the following properties:

* + - target The blackboard object about which the assumption

was made

* + - creator The knowledge source that created the assumption
    - reason The reason the knowledge source made the

assumption

* + - plainLetter The plaintext letter about which the assumption is

being made

* + - cipherLetter The assumed value of the plaintext letter

The need for each of these properties is largely derived from the very nature of an assumption: A particular knowledge source makes an assumption about a plain- text/ciphertext mapping and does so for a certain reason (usually because some rule was triggered). The need for the first member, target, is less obvious. We include it because of the problem of backtracking. If we ever have to reverse an assumption, we must notify all blackboard objects for which the assumption was originally made, so that they in turn can alert the knowledge sources they depend on (via the dependency mechanism) that their meaning has changed.

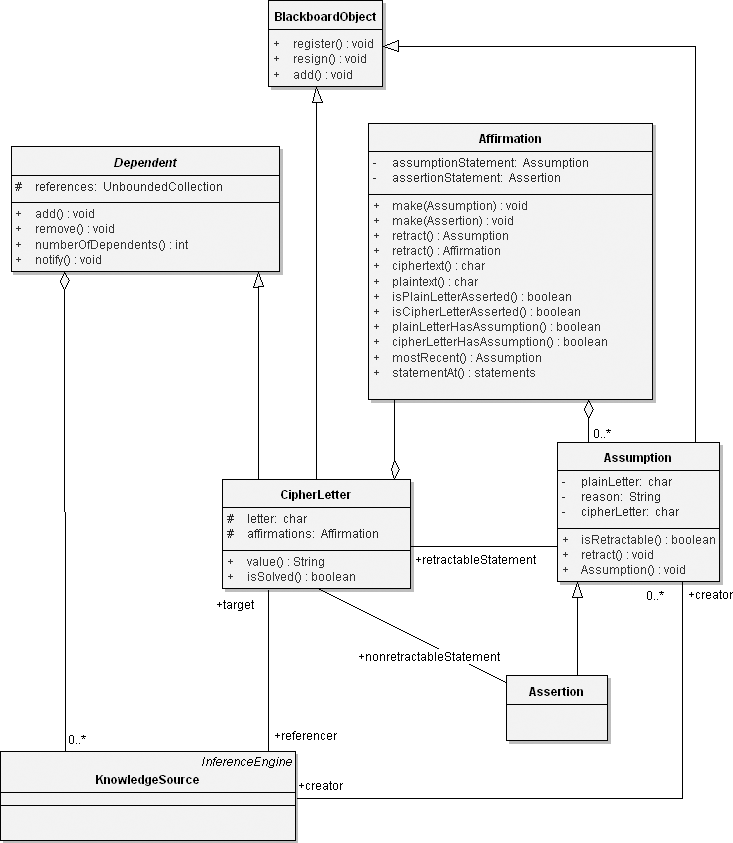
Next, we declare the subclass of Assumption named Assertion. The classes

Assumption and Assertion share the following operation, among others:

* + - isRetractable A selector: Is the mapping temporary?

All Assumption objects answer true to the predicate isRetractable, whereas all Assertion objects answer false. Additionally, once made, an asser- tion can neither be restated nor retracted.

Figure 10–3 provides a class diagram that illustrates the collaboration of the Dependent and Affirmation classes. Pay particular attention to the roles each abstraction plays in the various associations. For example, a Knowledge- Source is the creator of an Assumption and is also the referencer of a CipherLetter. Because a role represents a different view than an abstraction presents to the world, we would expect to see a different protocol between knowl- edge sources and assumptions than between knowledge sources and letters.



**Figure 10–3** Dependency and Affirmation Classes

# Construction

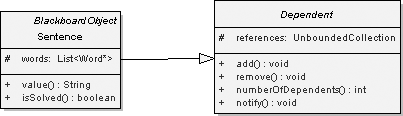
Let’s continue our design of the Sentence, Word, and CipherLetter

classes, followed by the Alphabet class, by doing a little isolated class design.

## Designing the Blackboard Objects

A sentence is quite simple: It is a BlackboardObject as well as a Dependent, and it denotes a list of words that compose the sentence.

We make the superclass Dependent abstract1 (Figure 10–4) because we expect there may be other Sentence subclasses that try to inherit from Dependent as well. By marking this inheritance relationship abstract, we cause such subclasses to share a single Dependent superclass.



**Figure 10–4** The Sentence Class Design with the Abstract Dependent Class

In addition to the operations register and resign defined by its superclass BlackboardObject, plus the four operations defined in Dependent, we add the following two sentence-specific operations:

* value Return the current value of the sentence.
* isSolved Return true if there is an assertion for all words in the

sentence.

At the start of the problem, value returns a string representing the original cryp- togram. Once isSolved evaluates as true, the operation value may be used to retrieve the plaintext solution. Accessing value before isSolved is true will yield partial solutions.

1. In UML 2.0, an abstract class is represented with the class name in italics. A keyword

{abstract} may also be placed in the property list.

Just like the Sentence class, a Word is a kind of BlackboardObject as well as a kind of Dependent. Furthermore, a Word denotes a list of letters. To assist the knowledge sources that manipulate words, we include a reference from a word to its sentence, as well as from a word to the previous and next words in the sentence.

As we did for the Sentence operations, we define the following two operations for the class Word:

* + value Return the current value of the word.
  + isSolved Return true if there is an assertion for every letter in the word.

We may next define the class CipherLetter. An instance of this class is a kind of BlackboardObject and a kind of Dependent. In addition to its inherited behaviors, each CipherLetter object has a value (such as the ciphertext letter H) together with a collection of assumptions and assertions regarding its corre- sponding plaintext letter. We can use the class Affirmation to collect these statements. Figure 10–5 illustrates the addition of the design of CipherLetter and Word in our architecture framework.

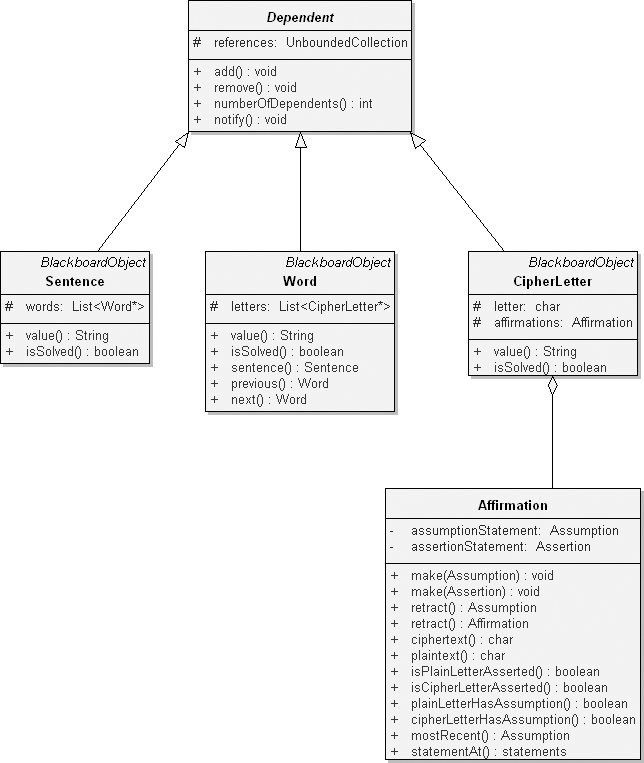
Notice that we include the selectors value and isSolved, similar to our design of Sentence and Word. We must also eventually provide operations for the clients of CipherLetter to access its assumptions and assertions in a safe manner.

One comment about the member object affirmations: We expect this to be a col- lection of assumptions and assertions ordered according to their time of creation, with the most recent statement in this collection representing the current assump- tion or assertion. The reason we choose to keep a history of all assumptions is to permit knowledge sources to look at earlier assumptions that were rejected, so that they can learn from earlier mistakes. This decision influences our design decisions about the Affirmation class, to which we add the following operations:

* + mostRecent A selector: returns the most recent assumption or assertion
  + statementAt A selector: returns the nth statement

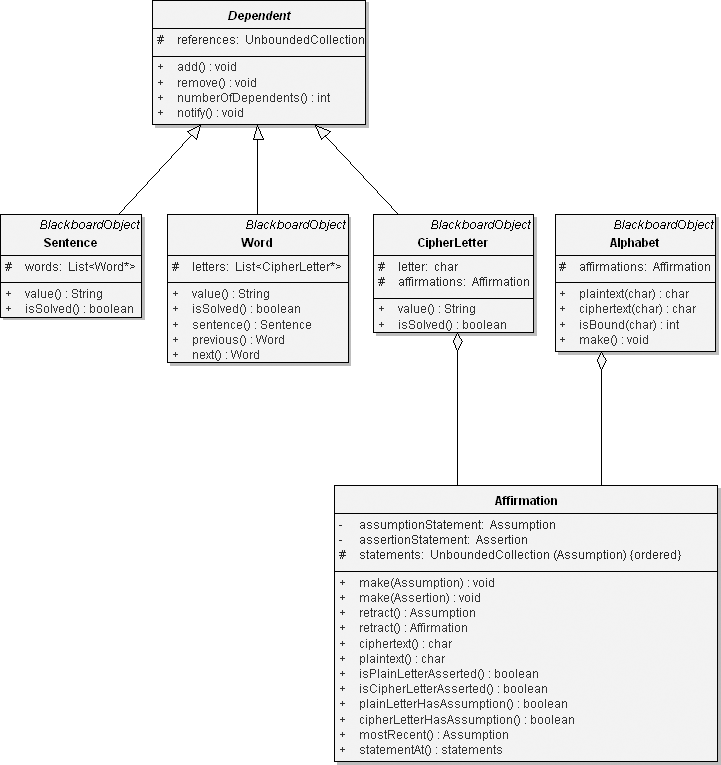
Now that we have refined its behavior, we can next make a reasonable implemen- tation decision about the Affirmation class. Specifically, we can include the protected member object statements, which is defined as a collection of assumptions.

Consider next the class named Alphabet. This class represents the entire plain- text and ciphertext alphabet, plus the mappings between the two. This informa- tion is important because each knowledge source can use it to determine which



**Figure 10–5** The Design of the CipherLetter and Word Classes

mappings have been made and which are yet to be done. For example, if we already have an assertion that the ciphertext letter C is really the letter M, then an alphabet object records this mapping so that no other knowledge source can apply the plaintext letter M. For efficiency, we need to query about the mapping both ways: Given a ciphertext letter, return its plaintext mapping, and given a plaintext letter, return its ciphertext mapping. Figure 10–6 illustrates the addition of the design of the Alphabet class.



**Figure 10–6** The Design of the Alphabet Class

Just as for the CipherLetter class, we also include a protected member object

affirmations and provide suitable operations to access its state.

Now we are ready to define the Blackboard class. This class has the simple responsibility of collecting instances of the BlackboardObject class and its subclasses. Thus we may design Blackboard as a type of instance of a DynamicCollection. We have chosen to inherit from rather than contain an

instance of the DynamicCollection class because Blackboard passes our test for inheritance: A Blackboard is indeed a kind of collection.

The Blackboard class provides operations such as add and remove, which it inherits from the Collection class. Our design includes five operations spe- cific to the blackboard.

* + reset Clean the blackboard.
  + assertProblem Place an initial problem on the blackboard.
  + connect Attach the knowledge source to the blackboard.
  + isSolved Return true if the sentence is solved.
  + retrieveSolution Return the solved plaintext sentence.

The second operation is needed to create a dependency between a blackboard and its knowledge sources.

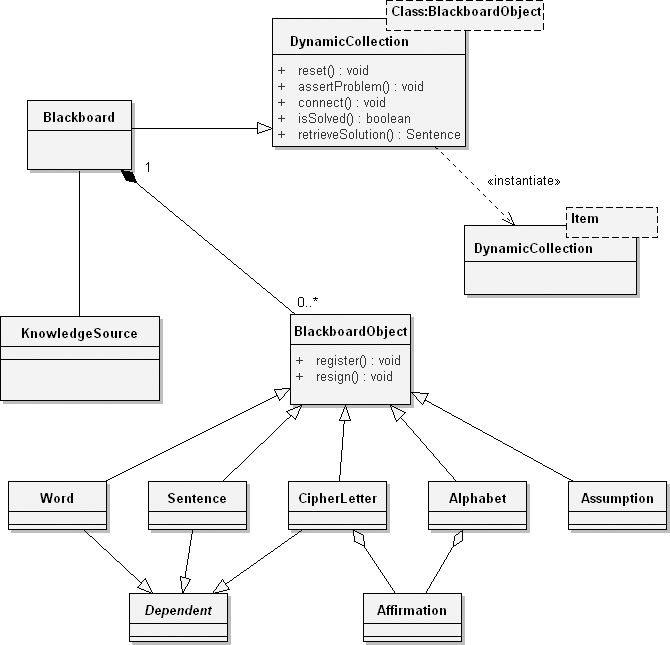
In Figure 10–7, we summarize our design of the classes that collaborate with Blackboard. In this diagram, notice that we show the Blackboard class as both instantiating and inheriting from the template class DynamicCollection. This diagram also clearly shows why introducing the Dependent class as an abstract class was a good design decision. Specifically, Dependent represents a behavior that encompasses only a partial set of BlackboardObject subclasses. By making Dependent abstract, we increase its chances of being reused.

## Designing the Knowledge Sources

In a previous section, we identified 13 knowledge sources relevant to this prob- lem. Just as we did for the Blackboard objects, we can design a class structure encompassing these knowledge sources and thereby elevate all common charac- teristics to more abstract classes.

### Designing Specialized Knowledge Sources

Assume for the moment the existence of an abstract class called Knowledge Source, whose purpose is much like that of the class BlackboardObject. Rather than treat each of the 13 knowledge sources as a direct subclass of this more general class, it is useful to first perform a domain analysis and see if there are any clusters of knowledge sources. Indeed, there are such groups: Some knowledge sources operate on whole sentences, others on whole words, others on contiguous strings of letters, and still others on individual letters. We may capture these design decisions by creating the following subclasses:



**Figure 10–7** The Refined Blackboard Class Diagram Design

* + SentenceKnowledgeSource Rules associated with sentences
  + WordKnowledgeSource Rules associated with words
  + LetterKnowledgeSource Rules associated with letters
  + StringKnowledgeSource Rules associated with strings

For each of these classes, we may provide another level of specification. For example, the subclasses of the class SentenceKnowledgeSource include the following:

* + SentenceStructureKnowledgeSource Rules specific to

sentence structure

* + SolvedKnowledgeSource Solved cryptogram sentence

Similarly, the subclasses of the intermediate class WordKnowledgeSource

include these:

* + WordStructureKnowledgeSource Rules specific to word

structure

* + SmallWordKnowledgeSource Rules specific to small

words

* + PatternMatchingKnowledgeSource Rules for matching

patterns of words

The last class requires some explanation. In our earlier list of the 13 knowledge sources, we said that the purpose of a pattern-matching knowledge source was to propose words that fit a certain pattern. We can use regular expression pattern- matching symbols such as:

* + Any item ?
  + Not item ~
  + Closure item \*
  + Start group {
  + Stop group }

With these symbols, we might give an instance of this class the pattern ?E~{A E I O U}, thereby asking it to give us from its dictionary all the three-letter words starting with any letter, followed by an E, and ending with any letter except a vowel. All instances of this class share a dictionary of words, and each instance has its own regular expression pattern-matching agent. The detailed behavior of this class is not important to us at this point in our design, so we will defer the invention of the remainder of its interface and implementation.

Continuing, we may declare the following subclasses of the class String KnowledgeSource:

* + CommonPrefixKnowledgeSource Rules specific to prefixes
  + CommonSuffixKnowledgeSource Rules specific to suffixes
  + DoubleLetterKnowledgeSource Rules for double letters, e.g.,

oo, ll, and so on

* + LegalStringKnowledgeSource Rules specific to what makes

legal strings

Lastly, we can introduce the following subclasses of the class

LetterKnowledgeSource:

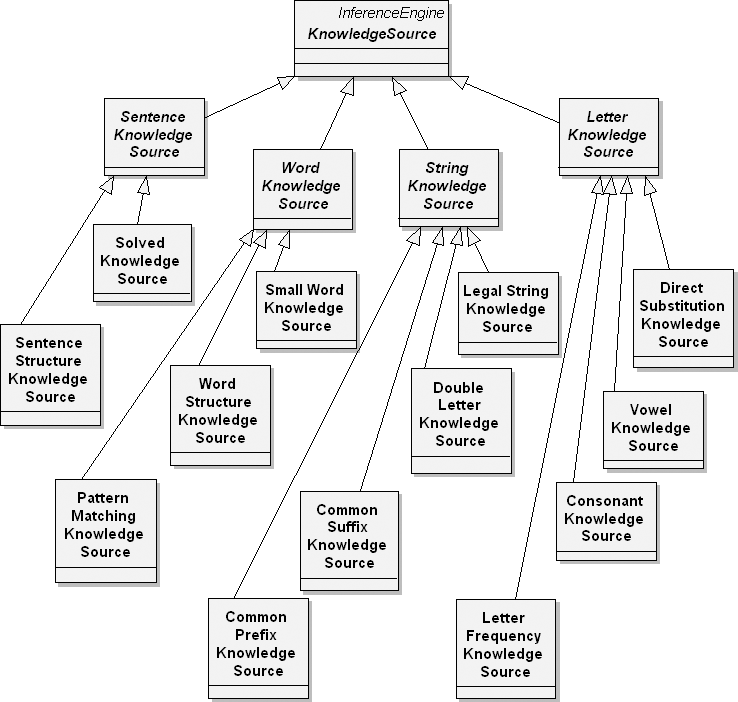
* + DirectSubstitutionKnowledgeSource Rules specific to

substitution of letters

* + VowelKnowledgeSource Rules specific for vowels
  + ConsonantKnowledgeSource Rules specific for

consonants

* + LetterFrequencyKnowledgeSource Rules specific to fre-

quency of letters Figure 10–8 illustrates the hierarchical structure of KnowledgeSource.

**Figure 10–8** The Generalization Hierarchy of the KnowledgeSource Class

### Generalizing the Knowledge Sources

Analysis suggests that only two primary operations apply to all these specialized classes:

* + reset Restart the knowledge source.
  + evaluate Evaluate the state of the blackboard.

The reason for this simple interface is that knowledge sources are relatively autonomous entities: We point one to an interesting Blackboard object and then tell it to evaluate its rules according to the current global state of the Blackboard. As part of the evaluation of its rules, a given knowledge source might do any one of several things.

* + Propose an assumption about the substitution cipher.
  + Discover a contradiction among previous assumptions, and cause the offending assumption to be retracted.
  + Propose an assertion about the substitution cipher.
  + Tell the controller that it has some interesting knowledge to contribute.

These are all general actions that are independent of the specific kind of knowl- edge source. To generalize even further, these actions represent the behavior of an inference engine. Therefore, we create the class InferenceEngine that, given a set of rules, evaluates those rules either to generate new rules (forward-chaining) or to prove some hypothesis (backward-chaining). When designing the construc- tor for InferenceEngine, the basic responsibility is to create an instance of this class and populate it with a set of rules, which it then uses for evaluation.

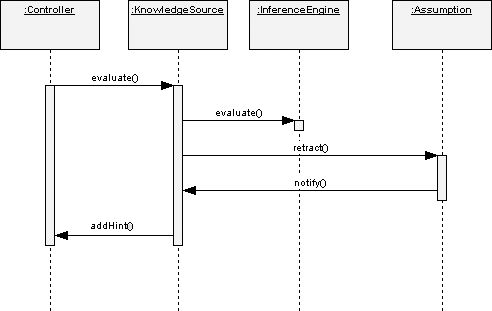
In fact, this class has only one critical operation that it makes visible to knowl- edge sources:

* + evaluate Evaluate the rules of the inference engine.

This then is how knowledge sources collaborate: Each specialized knowledge source defines its own knowledge-specific rules and delegates responsibility for evaluating these rules to the InferenceEngine class. More precisely, we may say that the operation KnowledgeSource::evaluate ultimately invokes the operation InferenceEngine::evaluate, the results of which are used to carry out any of the four actions we listed earlier. In Figure 10–9, we illustrate a common scenario of this collaboration.

The sequence diagram illustrates the following steps in this scenario:

1. Select a KnowledgeSource for action.
2. Evaluate the KnowledgeSource against the state of the Blackboard.



**Figure 10–9** A Scenario for Evaluating Knowledge Source Rules

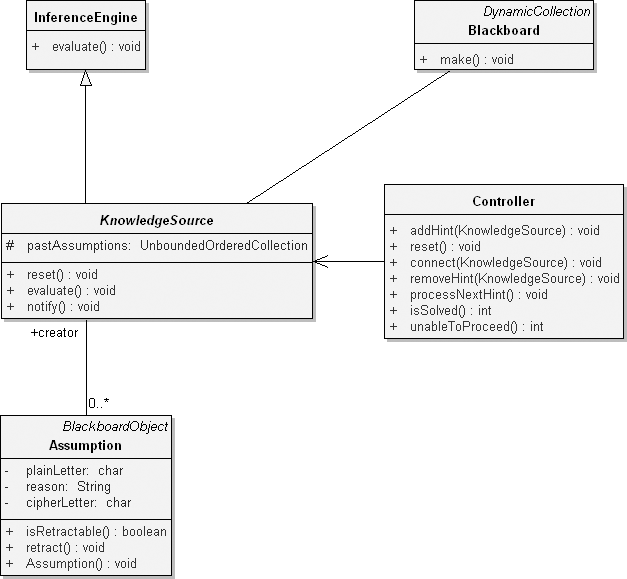
1. Take some action, such as retracting an Assumption.
2. Notify all dependent KnowledgeSource objects that the assumption has been retracted.
3. Tell the Controller that the KnowledgeSource has a new hint to offer in solving the blackboard problem.

What exactly is a rule? A rule might be composed for the common suffix knowl- edge source using a pattern-matching algorithm that recognizes a common suffix pattern such as \*I??. Given a string of letters matching the regular expression pattern \*I??, the candidate suffixes may include ING, IES, and IED.

In terms of its class structure, we may thus say that a knowledge source is a kind of inference engine. Additionally, each knowledge source must have some associ- ation with a blackboard object, for that is where it finds the objects on which it operates. Finally, each knowledge source must have an association to a controller, with which it collaborates by sending hints of solutions; in turn, the controller might trigger the knowledge source from time to time. Figure 10–10 illustrates these design decisions.

We also introduce the collection pastAssumptions, so that the knowledge source can keep track of all the assumptions and assertions it has ever made, in order to learn from its mistakes.

Instances of the Blackboard class serve as a repository of Blackboard objects. For a similar reason, we need a KnowledgeSources class, denoting the entire collection of knowledge sources for a particular problem.



**Figure 10–10** The Preliminary Design of KnowledgeSource

One of the responsibilities of this class is that when we create an instance of KnowledgeSources, we also create the 13 individual KnowledgeSource objects. We may perform three operations on instances of this class:

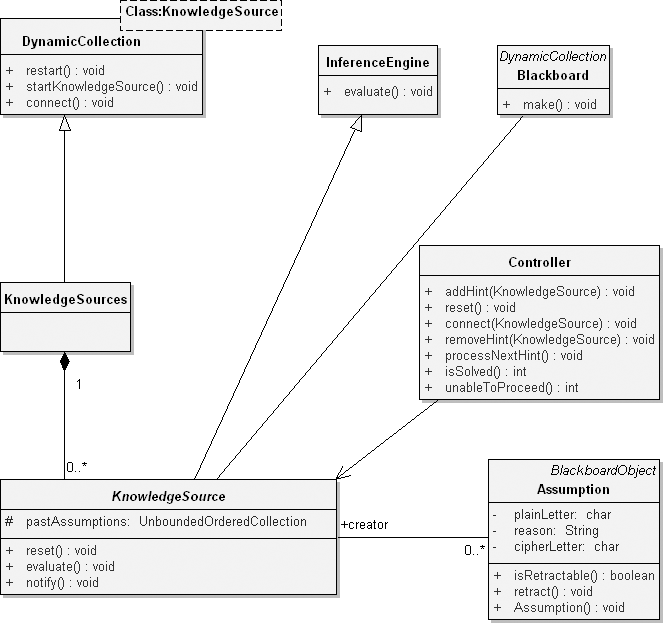
* + restart Restart the knowledge sources.
  + startKnowledgeSource Give a specific knowledge source its initial

conditions.

* + connect Attach the knowledge source to the black- board or to the controller.

Figure 10–11 provides the refined design of the class structure of the

KnowledgeSource classes, according to these design decisions.



**Figure 10–11** The Refined Design of the KnowledgeSource Class Diagram

## Designing the Controller

Consider for a moment how the controller and individual knowledge sources interact. At each stage in the solution of a cryptogram, a particular knowledge source might discover that it has a useful contribution to make and so gives a hint to the controller. Conversely, the knowledge source might decide that its earlier hint no longer applies and so may remove the hint. Once all knowledge sources have been given a chance, the controller selects the most promising hint and acti- vates the appropriate knowledge source by invoking its evaluate operation.

How does the controller decide which knowledge source to activate? We may devise a few suitable rules.

* + An Assertion has a higher priority than an Assumption.
  + The SolvedKnowledgeSource provides the most useful hints.
  + The PatternMatchingKnowledgeSource provides higher-priority hints than the SentenceStructureKnowledgeSource.

A controller thus acts as an agent responsible for mediating among the various knowledge sources that operate on a blackboard.

The controller must have an association to its knowledge sources, which it can access through the appropriately named class KnowledgeSources. Addition- ally, the controller must have as one of its properties a collection of hints, ordered in accordance with the above rules of prioritization. In this manner, the controller can easily select for activation the knowledge source with the most interesting hint to offer.

Engaging in a little more isolated class design, we offer the following operations for the Controller class.

* + reset Restart the controller.
  + addHint Add a knowledge source hint.
  + removeHint Remove a knowledge source hint.
  + processNextHint Evaluate the next-highest-priority hint.
  + isSolved A selector: Return true if the problem is solved.
  + unableToProceed A selector: Return true if the knowledge sources

are stuck.

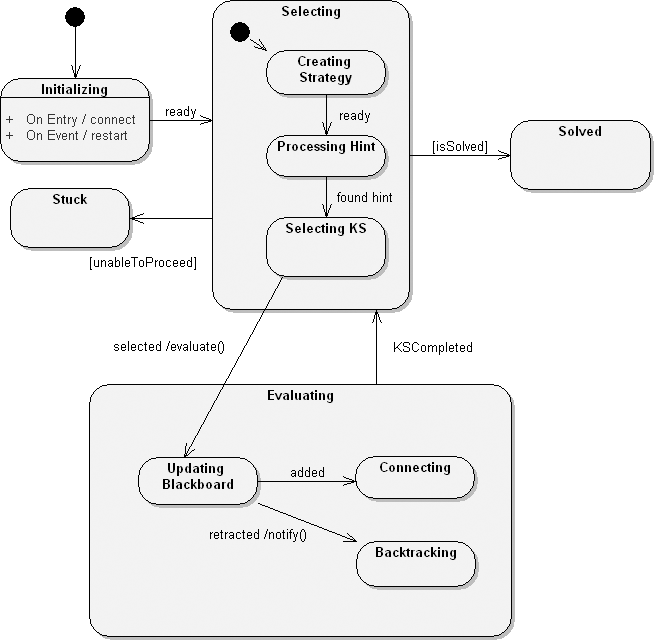
* + connect Attach the controller to the knowledge source.

The controller is in a sense driven by the hints it receives from various knowledge sources. As such, finite state machines are well suited for capturing the dynamic behavior of this class.

For example, consider the state transition diagram shown in Figure 10–12. Here we see that a controller may be in one of five major states: Initializing, Selecting, Evaluating, Stuck, and Solved. The controller’s most interesting activity occurs between the Selecting and Evaluating states. While selecting, the controller naturally transitions from the state Creating Strategy to Processing Hint and eventually to Selecting KS. If a knowledge source is in fact selected, then the controller transitions to the Evaluating state, wherein it first is in Updating Blackboard. It transi- tions to Connecting if objects are added and to Backtracking if assump- tions are retracted, at which time it also notifies all dependents.

The controller unconditionally transitions to Stuck if it cannot proceed and to

Solved if it finds a solved blackboard problem.



**Figure 10–12** The Controller State Machine

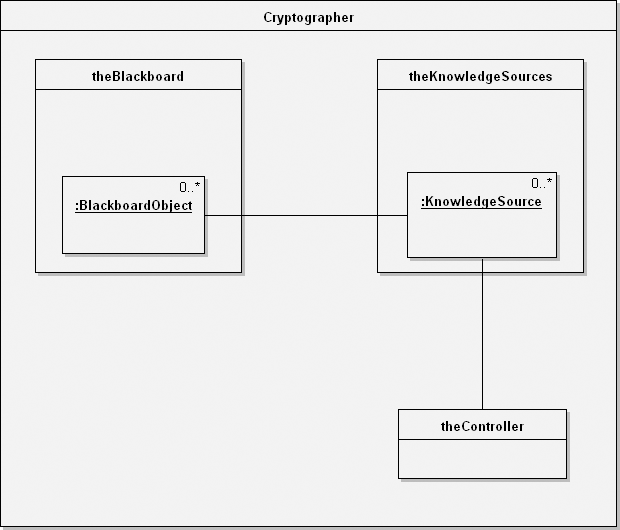
## Integrating the Blackboard Framework

Now that we have defined the key abstractions for our domain, we may continue by putting them together to form a complete application. We will proceed by implementing and testing a vertical slice through the architecture and then by completing the system one mechanism at a time.

### Integrating the Topmost Objects

Figure 10–13 is a composite structure diagram that captures our design of the top-

most object in the system, paralleling the structure of the generic blackboard



**Figure 10–13** The Cryptanalysis Composite Structure Diagram

framework shown earlier in Figure 10–1. In Figure 10–13, we show the physical containment of blackboard objects by the collection theBlackboard and knowledge sources by the collection theKnowledgeSources, using a short- hand style identical to that for showing nested classes.

In this diagram, we introduce an instance of a new class that we call Cryptographer. The intent of this class is to serve as an aggregate encom- passing the blackboard, the knowledge sources, and the controller. In this manner, our application might provide several instances of this class and thus have several blackboards running simultaneously.

We define two primary operations for the Cryptographer class:

* + reset Restart the blackboard.
  + decipher Solve the given cryptogram.

The behavior we require as part of this class’s constructor is to create the depen- dencies between the blackboard and its knowledge sources, as well as between

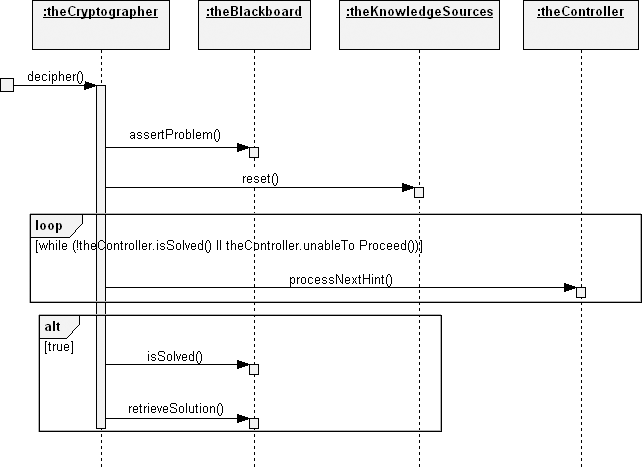
the knowledge sources and the controller. The reset method is similar, in that it simply resets these connections and returns the blackboard, the knowledge sources, and the controller back to a stable initial state.

Although we will not show its details here, the signature of the decipher oper- ation includes a string, through which we provide the ciphertext to be solved. In this manner, the root of our main program becomes embarrassingly simple, as is common in well-designed object-oriented systems.

The implementation of the decipher operation is, not surprisingly, slightly more complicated. Basically, we must first invoke the assertProblem opera- tion to set up the problem on the blackboard. Next, we must start the knowledge sources by bringing their attention to this new problem. Finally, we must loop, telling the controller to process the next hint at each new pass, either until the problem is solved or until all the knowledge sources are unable to proceed.

Figure 10–14 illustrates the flow of control using a sequence diagram.

We would be best advised to complete enough of the relevant architectural inter- faces so that we could complete this algorithm and execute it. Although at this point it would have minimal functionality, its implementation as a vertical slice



**Figure 10–14** The decipher Sequence Diagram

through the architecture would force us to validate certain key architectural

decisions.

Continuing, let’s look at two of the key operations used in decipher, namely, assertProblem and retrieveSolution. The assertProblem operation is particularly interesting because it must generate an entire set of Blackboard objects. In the form of a simple pseudocode script, our algorithm is as follows.

trim all leading and trailing blanks from the string return if the resulting string is empty

create a sentence object

add the sentence to the blackboard

create a word object (this will be the leftmost word in the sentence)

add the word to the blackboard add the word to the sentence

for each character in the string, from left to right if the character is a space

make the current word the previous word create a word object

add the word to the blackboard add the word to the sentence

else

create a cipher-letter object add the letter to the blackboard add the letter to the word

The purpose of design is simply to provide a blueprint for implementation. This script supplies a sufficiently detailed algorithm, so we need not show its complete implementation.

The operation retrieveSolution is far simpler; we simply return the value of the sentence on the blackboard. Calling retrieveSolution before isSolved evaluates true will yield partial solutions.

### Implementing the Assumption Mechanism

At this point, we have implemented the mechanisms that allow us to set and retrieve values for Blackboard objects. The next major function point involves the mechanism for making assumptions about Blackboard objects. This is a particularly significant issue because assumptions are dynamic (meaning that they are routinely created and destroyed during the process of forming a solu- tion), and their creation or retraction triggers controller events.

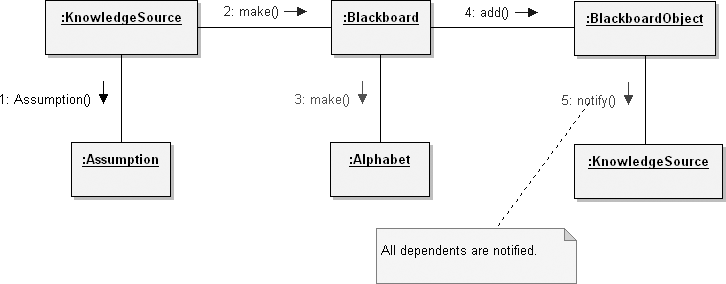
Figure 10–15 illustrates the primary scenario of when a knowledge source states an assumption. As this communication diagram shows, once the KnowledgeSource creates an Assumption, it notifies the Blackboard, which in turn makes the Assumption for its Alphabet and then for each BlackboardObject to which the Assumption applies. Using the depen- dency mechanism, the affected BlackboardObject in turn might notify any dependent KnowledgeSource.

In its most naive implementation, retracting an assumption simply undoes the work of this mechanism. For example, to retract an assumption about a cipher let- ter, we just pop its collection of assumptions, up to and including the assumption we are retracting. In this manner, the given assumption and all assumptions that built on it are undone.

A more sophisticated mechanism is possible. For example, suppose that we made an assumption that a certain one-letter word is really just the letter I (assuming we need a vowel). We might make a later assumption that a certain double-letter word is NN (assuming we need a consonant). If we then find we must retract the first assumption, we probably don’t have to retract the second one. This approach requires us to add a new behavior to the Assumption class so that it can keep track of what assumptions are dependent on others. We can reasonably defer this enhancement until much later in the evolution of this system because adding this behavior has no architectural impact.

## Adding New Knowledge Sources

Now that we have the key abstractions of the blackboard framework in place, and once the mechanisms for stating and retracting assumptions are working, our next



**Figure 10–15** The Assumption Mechanism

step is to implement the InferenceEngine class since all knowledge sources depend on it. As we mentioned earlier, this class has only one really interesting operation, namely, evaluate. We will not show its details here because this particular method reveals no new important design issues.

Once we are confident that our inference engine works properly, we may incre- mentally add each knowledge source. We emphasize the use of an incremental process for two reasons.

1. For a given knowledge source, it is not clear what rules are really important

until we apply them to real problems.

1. Debugging the knowledge base is far easier if we implement and test

smaller related sets of rules, rather than trying to test them all at once.

Fundamentally, implementing each knowledge source is largely a problem of knowledge engineering. For a given knowledge source, we must confer with an expert (perhaps a cryptologist) to decide which rules are meaningful. As we test each knowledge source, our analysis may reveal that certain rules are useless, oth- ers are either too specific or too general, and perhaps some are missing. We may then choose to alter the rules of a given knowledge source or even add new sources of knowledge.

As we implement each knowledge source, we may discover the existence of com- mon rules as well as common behavior. For example, we might notice that the WordStructureKnowledgeSource and the SentenceStructure KnowledgeSource classes share a common behavior, in that both must know how to evaluate rules regarding the legal ordering of certain constructs. The former knowledge source is interested in the arrangement of letters; the latter is interested in the arrangement of words. In either case, the processing is the same. Thus, it is reasonable for us to alter the knowledge source class structure by developing a new abstract class, called StructureKnowledgeSource, in which we place this common behavior.

This new knowledge source class hierarchy highlights the fact that evaluating a set of rules is dependent on both the kind of knowledge source as well as the kind of blackboard object. For example, given a specific knowledge source, it might use forward-chaining on one kind of Blackboard object and backward- chaining on another. Furthermore, given a specific Blackboard object, how

it is evaluated will depend on which knowledge source is applied.

# Post-Transition

In this section, we consider an improvement to the functionality of the cryptanal- ysis system and observe how our design weathers the change.

## System Enhancements

In any intelligent system, it is important to know what the final answer is to a problem, but it is often equally important to know how the system arrived at this solution. Thus, we desire our application to be introspective: It should keep track of when knowledge sources were activated, what assumptions were made and why, and so on, so that we can later question it, for example, about why it made an assumption, how it arrived at another assumption, and when a particular knowledge source was activated.

To add this new functionality, we need to do two things. First, we must devise a mechanism for keeping track of the work that the controller and each knowledge source perform, and second, we must modify the appropriate operations so that they record this information. Basically, the design calls for the knowledge sources and the controller to register what they did in some central repository.

Let’s start by inventing the classes needed to support this mechanism. First, we might define the class Action, which serves to record what a particular knowl- edge source or controller did. Figure 10–16 presents the design of the Action class as it fits into our architectural design.

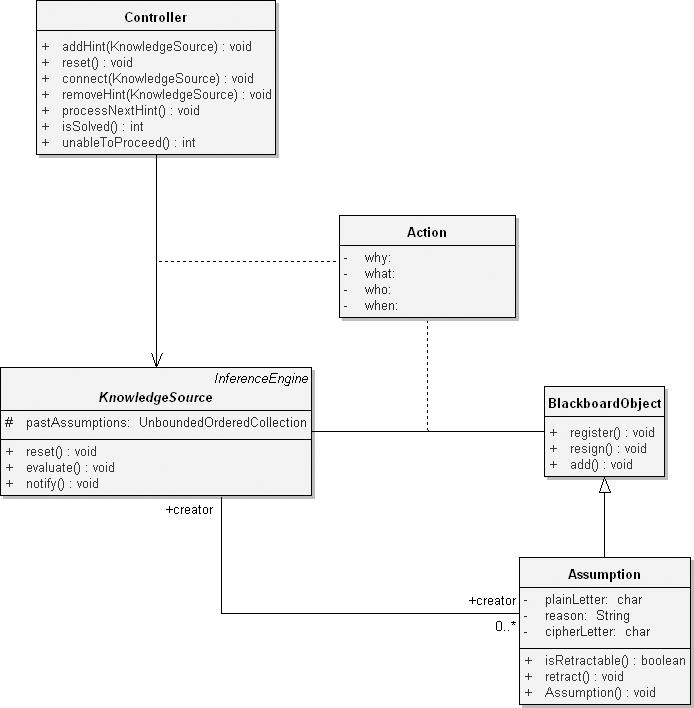
For example, if the controller selected a particular knowledge source for activa- tion, it would create an instance of this class, set the who argument to itself, set the what argument to the knowledge source, set the why argument to some explanation (perhaps including the current priority of the hint), and set when this occurred.

The first part of our task is done, and the second part is almost as easy. Consider for a moment where important events take place in our application. As it turns out, five primary kinds of operations are affected:

1. Methods that state an assumption
2. Methods that retract an assumption
3. Methods that activate a knowledge source
4. Methods that cause rules to be evaluated
5. Methods that register hints from a knowledge source

Actually, these events are largely constrained to two places in the architecture: as part of the controller’s finite state machine and as part of the assumption mecha- nism. Our maintenance task, therefore, involves touching all the methods that play a role in these two places, a task that is tedious but by no means rocket science. Indeed, the most important discovery is that adding this new behavior requires no significant architectural change.

To complete our work here, we must also implement a class that can answer who, what, when, and why questions from the user. The design of such an object is not terribly difficult because all the information it needs to know may be found as the state of instances of the class actions.



**Figure 10–16** Additional Functionality Provided through the Action Class Design

## Changing the Requirements

Once we have a stable implementation in place, many new requirements can be incorporated with minimal changes to our design. Let’s consider three kinds of new requirements:

1. The ability to decipher languages other than English
2. The ability to decipher using transposition ciphers as well as single substitu- tion ciphers
3. The ability to learn from experience

The first change is fairly easy because the fact that our application uses English is largely immaterial to our design. Assuming the same character set is used, it is mainly a matter of changing the rules associated with each knowledge source.

Actually, changing the character set is not that difficult either because even the

Alphabet class is not dependent on what characters it manipulates.

The second change is much harder, but it is still possible in the context of the blackboard framework. Basically, our approach is to add new sources of knowl- edge that embody information about transposition ciphers. Again, this change does not alter any existing key abstraction or mechanism in our design; rather, it involves the addition of new classes that use existing facilities, such as the InferenceEngine class and the assumption mechanism.

The third change is the hardest of all, mainly because machine learning is on the fringes of our knowledge in artificial intelligence. As one approach, when the controller discovers it can no longer proceed, it might ask the user for a hint. By recording this hint, along with the actions that led up to the system being stuck, the blackboard application can avoid a similar problem in the future. We can incorporate this simplistic learning mechanism without vastly altering any of our existing classes; as with all the other changes, this one can build on existing facilities.