**Artificial Intelligence: Cryptanalysis**

Artificial intelligence researchers tackle complex problems by constructing intelligent systems that emulate human behavior. Tasks like optimal route planning, navigating in low light, and filtering conversations at a party defy simple algorithmic solutions due to their complexity. These challenges prompt exploration into understanding human cognitive processes.

Intelligent systems exhibit several attributes that differentiate them from conventional systems:

- They adapt their goals over time.

- They possess, utilize, and update knowledge.

- They leverage diverse subsystems with various methods.

- They interact intelligently with users and other systems.

- They manage their own resources and attention.

Crafting intelligent systems is challenging due to the demanding nature of these properties. Moreover, when developing intelligent systems for critical domains like medical diagnosis or aircraft routing, ensuring safety is paramount as these systems may lack common sense knowledge.

Artificial intelligence research has yielded valuable concepts, such as knowledge representation and problem-solving architectures like rule-based expert systems and the blackboard model. In this chapter, we delve into designing an intelligent system for solving cryptograms using a blackboard framework, mimicking human problem-solving. Object-oriented development proves particularly suitable for this domain.

# **1 INCEPTION**

Our challenge lies in cryptanalysis, the endeavor to decrypt ciphertext into plaintext. While deciphering cryptograms is typically daunting, our task is relatively straightforward as we focus on single substitution ciphers.

## **1.1 Cryptanalysis Requirements**

Cryptography “embraces methods for rendering data unintelligible to unauthorized 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.

Deciphering a substitution cipher involves more than simple mapping; it often requires trial and error, starting with common patterns like one- and two-letter words to deduce the mapping of other letters. For example, deciphering "o" in a three-letter word might suggest possibilities like "one," "our," or "off," helping to unravel the rest of the message.

Knowledge of spelling, grammar, and common word patterns can aid in deciphering substitution ciphers. For instance, unlikely letter combinations like "qq" can be identified, and word endings such as "ing" can be inferred. Higher-level linguistic rules, like the likelihood of certain word sequences or the structure of sentences, can guide decryption efforts. For instance, recognizing common sentence structures can help identify missing components like nouns or agents.

The decryption process may involve backtracking when initial assumptions lead to contradictions or dead ends. For example, if assuming a certain two-letter word is "or" creates inconsistencies, alternatives like "of" or "on" must be explored, potentially requiring the reconsideration of earlier assumptions. Overall, our system must decipher a cryptogram back to its original plaintext, under the assumption of a simple substitution cipher.

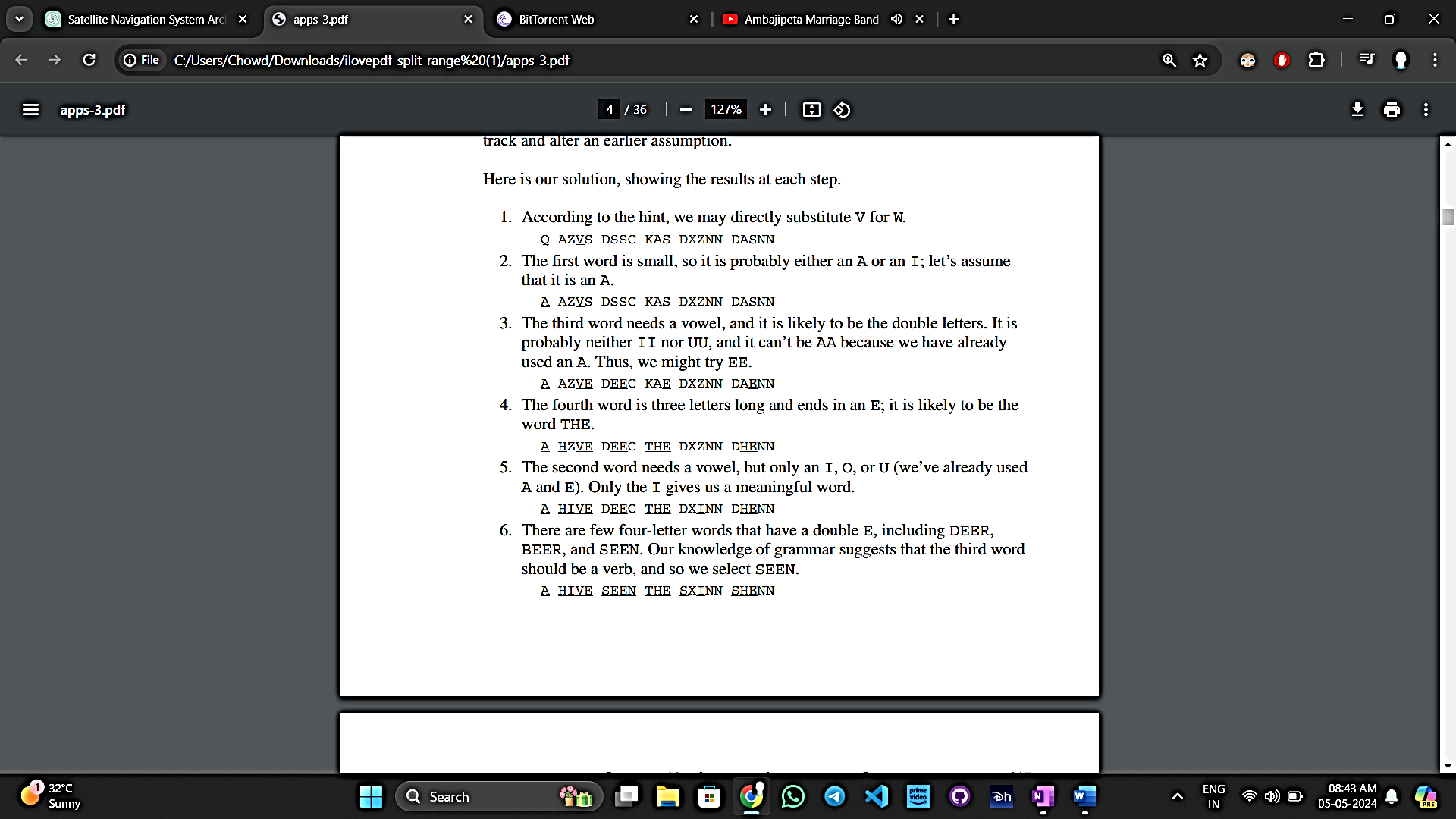
## **1.2 Defining the Boundaries of the Problem**

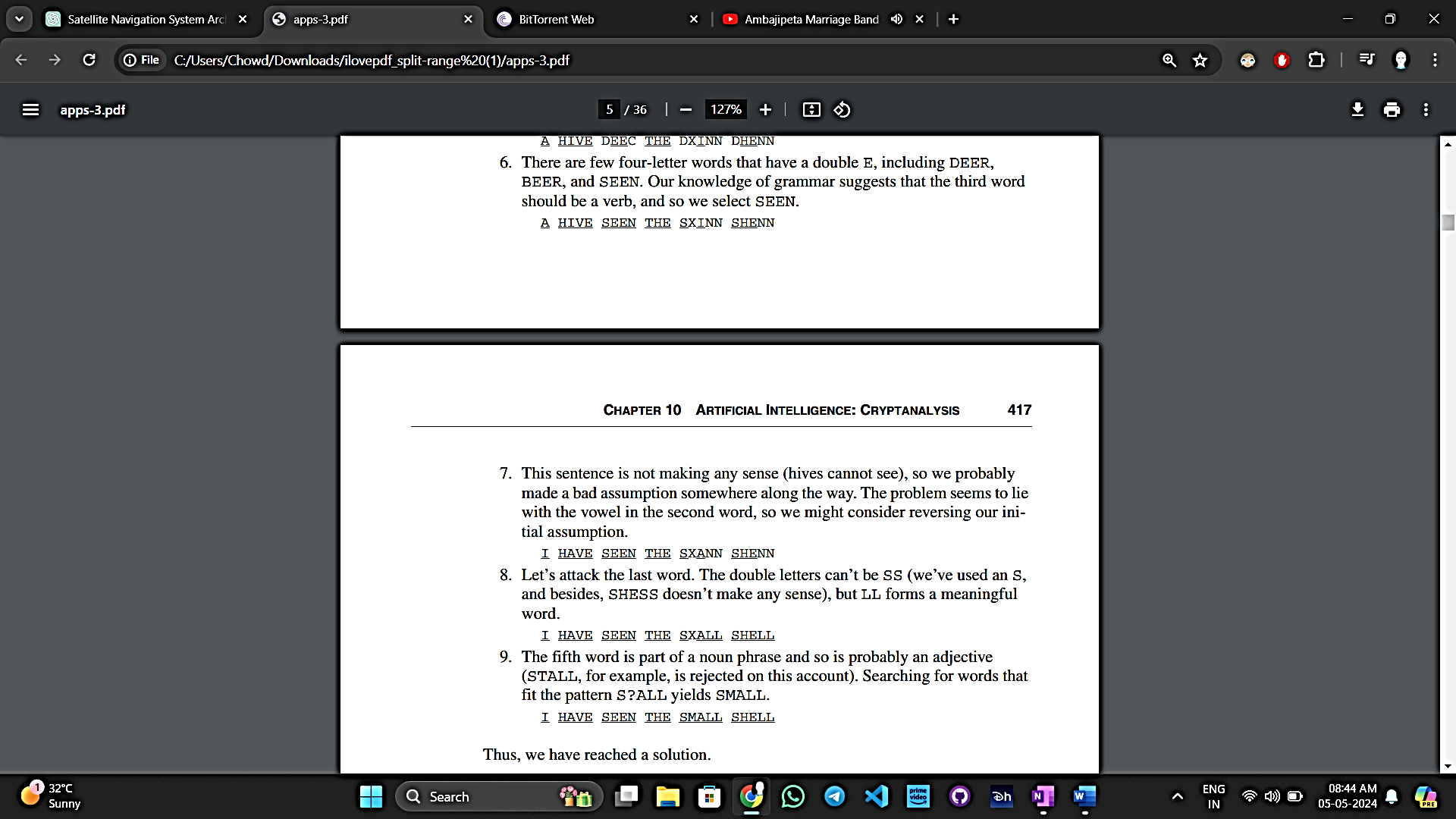
To solve the cryptogram "Q AZWS DSSC KAS DXZNN DASNN" based on the given hint that "W" represents "V", I'll start by identifying repeating patterns, common words, and frequently occurring letters. Then, I'll use my knowledge of English language structure and common letter combinations to make educated guesses about the substitutions.

Given the hint, I'll replace every occurrence of "W" in the cryptogram with "V". This might reveal common words or patterns. Then, I'll look for other repeating letters or sequences that could represent common words or combinations.

Continuing this process iteratively, I'll make assumptions, evaluate their implications, and backtrack if necessary. I'll also consider grammar and syntax rules to refine my substitutions and piece together the plaintext message.

Let's proceed step by step and see how it goes!





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

* **Using different types of knowledge:** We used what we knew about words, grammar, and spelling to solve the puzzle.
* **Keeping track of guesses:** We wrote down our guesses in one place and used our knowledge to see if they made sense.
* **“1”Thinking flexibly:** Sometimes we started with general rules and applied them to specific cases. Other times, we looked at specific clues to figure out broader patterns.
* “1”We reasoned opportunistically. At times, we reasoned from general to specific 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).

In simpler terms, a blackboard model is a way of solving complex problems by breaking them down into smaller parts and using different pieces of knowledge to find a solution. It's like having a shared space (the blackboard) where you write down guesses and ideas, and then use those to work towards solving the problem. This approach has been successfully used in various fields, like speech recognition and image understanding, because it's efficient and allows for collaboration between different pieces of knowledge.

## **1.3 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:

In essence, the blackboard model works like a group of people solving a jigsaw puzzle together. Imagine a room with a big blackboard and a group of people holding puzzle pieces. Each person examines their pieces and looks for ones that fit with what's already on the blackboard. When they find a match, they add their piece to the blackboard, which might trigger other pieces to fit as well. There's no need for direct communication because everyone knows when their piece will help solve the puzzle. It's a flexible and opportunistic approach, where the solution builds gradually as people add their pieces, rather than following a rigid, predetermined order.

The blackboard framework comprises three main components: the blackboard itself, multiple knowledge sources, and a controller that manages interactions among these sources. The blackboard serves as a repository for data essential for solving the problem, holding both input and output information. It contains objects representing elements of the solution space, organized into hierarchical levels of analysis. These objects and their properties define the language used to discuss and manipulate solutions.

Englemore and Morgan describe how the domain knowledge required to solve a problem is divided into separate and independent knowledge sources. Each knowledge source aims to provide information that contributes to solving the problem. When activated, a knowledge source accesses the current information on the blackboard and updates it according to its specialized knowledge. These knowledge sources are typically represented as procedures, sets of rules, or logic assertions.



“1”Knowledge sources, abbreviated as KSs, are tailored to specific domains. In speech recognition systems, they might include agents that understand phonemes, words, and sentences. Similarly, in image recognition systems, KSs would encompass agents knowledgeable about basic image elements like edges and textures, as well as higher-level concepts like objects in a scene such as buildings, roads, and people. Typically, KSs mirror the hierarchical structure of objects on the blackboard. Each KS takes input from one level, processes it, and produces or modifies output at another level. For example, a KS focusing on words in a speech recognition system might analyze phonemes to form complete words.

There are two main reasoning approaches in KSs: forward-chaining and backward-chaining. Forward-chaining involves starting with specific assertions and deriving a general conclusion, while backward-chaining begins with a hypothesis and tries to validate it using existing assertions. Control in the blackboard model is opportunistic because depending on the situation, a KS may employ either forward or backward chaining.

“1”Knowledge sources, or KSs, are specialized for specific domains like speech or image recognition. They match the hierarchical structure of objects on the blackboard, taking input from one level and producing or modifying output at another. For example, a KS for speech recognition might analyze phonemes to form complete words.

There are two reasoning approaches: forward-chaining, starting with specific assertions to derive a conclusion, and backward-chaining, beginning with a hypothesis and trying to validate it. Control in the blackboard model is opportunistic, allowing KSs to use either approach based on the situation.

Knowledge sources have preconditions and actions. Preconditions define the blackboard state the knowledge source is interested in, like identifying a linear region in image recognition. When a precondition is met, the knowledge source focuses on that part of the blackboard and takes action using its rules or procedural knowledge.

There's no need for constant polling; instead, when a knowledge source has something to contribute, it notifies the blackboard controller. The controller then selects the most promising knowledge source from those available.

## **1.4 Analysis of Knowledge Sources**

To analyze knowledge sources for our cryptogram-solving problem, we'll follow a typical approach used in knowledge engineering. This involves collaborating with a domain expert to document their problem-solving heuristics. We can achieve this by solving several cryptograms while documenting the thought process.

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 fourletter 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, conjunctives, and so on |

From an object-oriented perspective, each of these 13 knowledge sources can be seen as a potential class in our system architecture. Each class instance holds specific state (its knowledge), exhibits behavior unique to its class (like how a suffix knowledge source reacts to words with common endings), and has its own identity (such as the existence of a small-word knowledge source independent of pattern-matching knowledge sources).

These knowledge sources can also be organized into a hierarchy based on their scope of operation. Some work at the sentence level, others at the letter level, and some handle groups or individual letters. This hierarchy mirrors the objects that might be present on the blackboard: sentences, words, strings of letters, and individual letters.

# **2 ELABORATION**

We're now poised to devise a solution to the cryptanalysis challenge by leveraging the blackboard framework we've outlined. This exemplifies a prime case of large-scale reuse, as we employ a tried-and-tested architectural pattern as the basis of our design.

Based on the blackboard framework architecture, our system's top-level entities include a blackboard, multiple knowledge sources, and a controller. Our immediate objective is to pinpoint the domain-specific classes and objects that refine these overarching abstractions.

## **2.1 Blackboard Objects**

“1”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 abstractions of the knowledge sources. The knowledge sources use the blackboard as a global source of input data, partial solutions, alternatives, final solutions, and control information.

“1”The blackboard is like a big bulletin board with different layers, each representing a level of detail. These layers match up with the various levels of complexity in our problem-solving strategies. The knowledge sources check this board for information they need to solve the puzzle, like hints, partial answers, and instructions on what to do next.

“2”To begin the design of the blackboard’s hierarchical structure, we identify the following 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 alphabet 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

“2”we're setting up the structure of our blackboard by defining different types of information that will be stored on it. We start by recognizing that our cryptogram can be broken down into sentences, words, and individual letters. Additionally, we need to keep track of the assumptions made by our knowledge sources, like which letters they've guessed for the plaintext and ciphertext. Finally, we'll have a class to manage the relationship between the plaintext and ciphertext alphabets.

There's a commonality among these classes: they all represent objects that can be stored on the blackboard. To capture this shared characteristic, we create a superclass called BlackboardObject. This superclass serves as the foundation for all objects that can be placed on the blackboard.

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

We define "register" and "resign" as operations on instances of BlackboardObject rather than on the Blackboard itself because the BlackboardObject instances have the detailed knowledge and responsibility to manage their own attachment to or removal from the Blackboard. Each BlackboardObject needs to be self-aware when it's attached to the Blackboard so it can participate effectively in problem-solving on the Blackboard. This responsibility aligns well with the behavior and knowledge encapsulated within each BlackboardObject instance.



## **2.2 Dependencies and Affirmations**

The introduction of the abstract class "Dependent" provides a mechanism for individual sentences, words, and cipher letters to maintain references to knowledge sources that depend on them. This ensures that when there's a change in an assumption about an object, the relevant knowledge sources can be notified. By implementing this class, objects in our system can manage their dependencies effectively, facilitating communication and coordination with associated knowledge sources.

To design the Dependent class, we include an object that represents a collection 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" functions as a passive iterator, allowing us to perform a specified operation on every dependent object when invoked. This flexibility enables efficient handling of dependencies across the collection.

The "Dependency" property can be independently mixed into other classes, enhancing their functionality. For instance, a "CipherLetter" can be both a "BlackboardObject" and a "Dependent," enabling seamless integration of these abstractions for desired behavior. Utilizing an abstract class in this manner promotes reusability and enhances the separation of concerns within the architecture.

Additionally, both "CipherLetter" and "Alphabet" share another common property: instances of these classes may have assumptions made about them, managed by the "Assumption" object, which is also a type of "BlackboardObject." To track assumptions and assertions about associated objects, we introduce the "Affirmation" class.

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.

To enhance our model, we'll distinguish between assumptions and assertions. An assumption represents a temporary mapping between a ciphertext and its corresponding plaintext, while an assertion denotes a permanent mapping. As the cryptogram solution progresses, assumptions may evolve into assertions. To accommodate this, we refine the Assumption class and introduce a new subclass called Assertion, both managed by instances of the Affirmation class and placed on the blackboard.

Assumption objects, being of general interest to all knowledge sources, inherit from BlackboardObject. They hold properties such as the target (the associated blackboard object), creator (the knowledge source), reason, plainLetter, and cipherLetter. These properties reflect the nature of an assumption, including its origin and the mapped letters.

Assertion, a subclass of Assumption, shares operations like isRetractable. While Assumptions can be retracted, Assertions cannot, as they represent definitive mappings.

Figure 10–3 depicts the collaboration between Dependent and Affirmation classes. For example, a KnowledgeSource creates an Assumption and refers to a CipherLetter. Different associations exist between knowledge sources and assumptions, and between knowledge sources and letters, highlighting distinct roles and protocols.



# **3 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.

In designing the Sentence, Word, and CipherLetter classes, as well as the Alphabet class, we'll start with isolated class design. This involves outlining the properties and behaviors of each class independently before considering their interactions within the broader system.

## **3.1 Designing the Blackboard Objects**

Designing the Blackboard Objects involves defining the properties and behaviors of each class, such as Sentence, Word, and CipherLetter.

For instance, a Sentence can be described as a BlackboardObject and a Dependent, indicating its presence on the blackboard and its dependency on other objects. It typically comprises a list of words that form the sentence.

To facilitate potential subclassing, the superclass Dependent is made abstract, allowing for shared behavior among subclasses. This abstraction ensures consistency in handling dependencies across different subclasses inheriting from Dependent.



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 cryptogram. 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.

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.

For the CipherLetter class, we define it as a subclass of BlackboardObject and Dependent. Each CipherLetter object holds a value (e.g., the ciphertext letter "H") and a collection of assumptions and assertions regarding its corresponding plaintext letter. These statements are managed by an Affirmation object.

The Affirmation class maintains a history of assumptions and assertions, allowing knowledge sources to learn from past mistakes. It includes operations like mostRecent and statementAt to access these statements.

As for the Alphabet class, it represents the plaintext and ciphertext alphabets along with their mappings. This information is crucial for knowledge sources to determine potential plaintext-ciphertext mappings.



# **4.TRANSISTION**