Case Studies of Several Case-Based Reasoners

In this chapter, we present case studies of several case-based reasoners. Through this presentation, readers should begin to understand better the types of tasks case-based reasoning can be used for and the types of input, knowledge, and reasoning architectures that go into making a case-based reasoner run. We present six automated reasoners:

1. CHEF as a case-based planner
2. JULIA as a case-based designer
3. CASEY as a case-based diagnosis program
4. HYPO as a case-based interpretive program
5. PROTOS as a case-based classification program
6. CLAVIER as a case-based program that is in use and saving its company money

We then discuss several hypothetical and real “retrieval-only” case-based systems. These are interactive systems that help a person perform a reasoning task or that provide advice.

For each system, we discuss its domain and task, the knowledge it uses, its input, its architecture, and the reasoning it does. The representations, knowledge, and reasoning used in these programs will be discussed in far more detail in later chapters of this book as we discuss the ways in which each of the components of a case-based reasoner can be implemented. Note that each system presented was chosen because it is representative of a class of case-based systems. Thus, some systems whose pieces are discussed extensively later in the book (e.g., MEDIATOR) are not presented in detail in this chapter. Readers can find out more about those systems by consulting the Appendix.
2.1 CHEF

CHEF is a case-based planner (Hammond 1986a, 1986b, 1989a). It takes as input a conjunc-
tion of subgoals that it needs to plan to achieve, and its output is a plan. Its domain is recipe
creation. Recipes are viewed, in CHEF, as plans. They provide the sequence of steps that must
be carried out to achieve the creation of some dish. Thus, CHEF’s input is goals that recipes
can achieve (e.g., include fish, use stir-frying method, achieve savory taste); its output is a rec-
ipe (plan) that can achieve those goals.

As a case-based planner, CHEF creates its plans by recalling old plans that worked
under similar circumstances and modifying them to fit its new situation. Thus, its first step in
creating a plan is to retrieve an old recipe that fulfills as many of its new goals as possible. To
canonicalize this sort, CHEF indexes its plans by the goals they achieve. Beef and
broccoli, for example, is indexed by several goals, among them include beef, include a
coarse vegetable, use method stir-fry, and achieve taste savory.

Next, CHEF adapts the old plan to fit the new situation. It does this in two steps. First, it
reinstatates the old plan. That is, it creates an instance of it that substitutes new objects for
the ones used previously. If it is creating a chicken and snow peas recipe, for example, from its
beef and broccoli recipe, it substitutes chicken for beef and snow peas for broccoli in the old
recipe. To do this, CHEF needs to know something about the roles its objects play in the old
plans. CHEF has fairly limited knowledge of this, and the particular method it uses to know
what should be substituted for what is to look at the similarities between the objects in the old
and new situations and substitute those items in the new situation that are most similar to
objects in the old one. Because chicken and beef are both defined as meat, for example, it
decides that they correspond; similarly for broccoli and snow peas—both are vegetables.

In its second adaption step, CHEF applies special-purpose object critics to modify the
old plan for the new situation. A typical one says that duck must be defatted before stir-frying.

It looks something like this:

After doing step: bone the duck
    do: clean the fat from the duck
because: the duck is now fatty

This critic is associated with the object duck, and each time it is used in a recipe, the critic is
fired off. If there is a step of boning the duck in the recipe being created, a step is added after
that one specifying that the duck should be defatted.

Object critics most often add special-purpose preparation steps to a plan (e.g., deveining
shrimp, chopping items into pieces of the right size, deboning, defatting). They are CHEF’s
way of encoding knowledge about special procedures associated with the use of objects within
its domain. Their use during adaptation shows the interplay between the use of experience and
general knowledge in a case-based reasoning system.

After reinstatement and application of critics, CHEF has a complete plan. Many plan-
ners would stop there and be finished. Remember, though, that a hallmark of case-based rea-
soning is that a reasoner learns from its experiences. If the planner stopped with creation of its
plan, it would have no feedback to tell it whether its plan worked or didn’t work. Without that
feedback, there is little basis for learning. The reasoner could store the new plans it created,
but it would have no way of knowing later on whether it was repeating a plan that worked or one that didn't. In addition, in complex situations, it often isn't possible to predict what will and will not work with certainty. Only trying plans out in the complexity of the real world can provide that insight.

CHEF therefore goes on to use its plan and collect feedback about how it worked. When a plan works well, it is finished. It stores the plan in memory and goes on to its next task. When a plan doesn't work as expected, however, CHEF attempts to learn from the situation. It creates a causal explanation of why the plan didn't work and uses that to index into its general planning knowledge, which it uses to repair the faulty plan.

CHEF tries its plans by running them in a simulator that provides feedback much like that the world would supply (to the extent that the world is captured in the simulator). When it runs its original recipe for beef and broccoli through the simulator, for example, it finds that the broccoli turns soggy. It recognizes this as a failure of the plan and attempts to explain why it happened. In this case, it finds that extra liquid in the pan caused the broccoli to cook too much and that the extra liquid is a side effect of a previous step and does not directly achieve any goal. CHEF thus classifies this failure as a side-effect: disabled-condition: concurrent. That is, a side effect of one step disabled a condition of another step, and the side effect and disabled condition happen concurrently with each other. Associated with this sort of failure, CHEF has three repair strategies: splitting and re-forming steps in the plan, altering the plan to get rid of the side effect, and finding an alternative plan. It chooses to split and re-form.

We need to consider now how CHEF can recognize the failures of its plans. The answer is that after CHEF creates a plan, it does the equivalent of trying it out in the world (by running it in its simulator). This feedback enables it to notice whether its plans were successful or not by comparing real results (from the simulator) to expected results. A program might also recognize failures by projecting the results of its plans as well as it can, predicting their results, and then comparing those projections to its expectations.

Another thing we should consider is the classifications CHEF uses to categorize its failures. As can be seen from the one used in this case, CHEF's explanations are in terms of causal relationships between goals, plans, and steps of plans that provide a general vocabulary for describing general planning situations. These situation descriptions function similarly to the critics in HACKER (Sussman 1975) and NOAH (Sacerdotti 1977), though they are more flexible than those critics. Each describes a general plan failure situation and points to a variety of strategies for repairing that sort of failure. The major difference between these structures, called TOP (Schank 1982; Hammond 1984), and the critics in NOAH and HACKER is that TOP organize information about these sorts of situations, whereas critics are rules that associate one repair with each failure type.

CHEF chooses a repair strategy by first looking to see if there are any similar cases stored in the TOP that suggest a repair and then checking the suitability conditions for each repair plan to see which is most appropriate. It applies an appropriate repair plan and fixes its faulty plan.

CHEF continues with one more step before it is finished. It updates its understanding of the world so that it will be able to anticipate and not repeat the mistake it has just made. To do that, it figures out what the predictors of this type of failure are and uses those as indexes to a warning that a failure might happen. It chooses its indexes by examining the previously
FIGURE 2.1 CHEF’s Functional Architecture

derived explanation of this failure, extracting from it those parts that describe environmental features responsible for the failure. It uses this index to index both the case with the failed plan, which acts as a warning case, and the case with the repaired plan, which provides it a way to plan around that particular problem. For the beef and broccoli recipe, the best predictor is that the vegetable being stir-fried needs to remain crisp. This becomes an index to the faulty beef and broccoli recipe, which warns of the potential for sogginess and for the repaired beef and broccoli recipe, which provides a way to keep the vegetable crisp.

After indexing the new plan by the effects it avoids, CHEF is able to anticipate, and thereby avoid, making the same mistake in the future. It does this by inserting one additional
step in its planning process before plan creation: anticipation of failures. In this step, which is its first, it looks specifically for plans with goals similar to its current ones that failed, and it adds to the description of its current situation that it should avoid such failures.

Figure 2.1 shows the pieces of CHEF's architecture. Rectangles represent its functional units, circles represent its knowledge sources, and ovals represent the input and output of each functional process. A portion of CHEF's output, shown in Figures 2.2 and 2.3, traces the steps CHEF uses in creating and correcting the beef and broccoli recipe. Figure 2.4 shows how CHEF uses what it has learned from that experience to anticipate and avoid the same failure in making chicken and snow peas at a later date.

---

**RETRIEVER:**
Searching for plan that satisfies--
  - Include beef in the dish.
  - Include broccoli in the dish.
  - Make a stir-fry dish.

Found recipe -> RECIPE BEEF-WITH-GREEN-BEANS

Recipe exactly satisfies goals ->
  - Make a stir-fry dish.
  - Include beef in the dish.

Recipe partially matches ->
  - Include broccoli in the dish.
  - in that the recipe satisfies:
    - Include vegetables in the dish.

**MODIFIER:**
Building new name for copy of BEEF-WITH-GREEN-BEANS
Calling recipe BEEF-AND-BROCCOLI

Modifying recipe: BEEF-AND-BROCCOLI
to satisfy: Include broccoli in the dish.

Placing some broccoli in recipe BEEF-AND-BROCCOLI

---Considering ingredient-critic:
  - Before doing step: Stir-fry the -Variable-
    - do: Chop the broccoli into pieces the size of chunks.
  --ingredient-critic applied.

Projected results:
  - The beef is now tender.
  - The dish now tastes salty.
  - The dish now tastes savory.
  - The dish now tastes sweet.
  - The broccoli is now crisp.
  - The dish now tastes like garlic.

**SIMULATOR:**
Executing recipe.

**FIGURE 2.2** CHEF's Steps in Creating Beef and Broccoli
REPAIRER:
Checking results.

Checking goals of recipe -> BEEF-AND-BROCCOLI ...

The goal: The broccoli is now crisp is not satisfied.
It is instead the case that: The broccoli is now soggy.

Changing name of recipe BEEF-AND-BROCCOLI
to BAD-BEEF-AND-BROCCOLI

Explaining the failures ...

ANSWER: The step: Stir-fry the sugar, soy sauce, rice wine, garlic,
cornstarch, broccoli, and beef for three minutes enables the
satisfaction of the following goals:
  The dish now tastes savory.
The beef is now tender.
But the same step results in thin liquid in the pan from the
beef, resulting in the failure

Found TOP TOPI -> SIDE-EFFECT:DISABLED-CONDITION:CONCURRENT
  It has 3 repair strategies associated with it:
    SPLIT-AND-REFORM
    ALTER-PLAN:SIDE-EFFECT
    ADJUNCT-PLAN

Applying TO8 to failure in recipe BAD-BEEF-AND-BROCCOLI
Asking questions needed for evaluating strategy:
   SPLIT-AND-REFORM ...

Implementing plan -> Instead of doing step: Stir-fry the sugar, soy sauce,
rice wine, garlic, cornstarch, broccoli, and beef for 3 minutes
Do: S1 - Stir-fry the broccoli for 3 minutes.
S2 - Remove the broccoli from the result of action S1.
S3 - Stir-fry the sugar, soy sauce, rice wine, garlic, cornstarch, and
  beef for 3 minutes.
S4 - Add the result of action S2 to the result of action S3.
S5 - Stir-fry the result of action S4 for a half minute.
Suggested by strategy SPLIT-AND-REFORM.

ASSIGNER:
Building demons to anticipate failure.

Building demons: DEMON6 to anticipate interaction between rules:
"Meat sweats when it is stir-fried."
"Stir-frying in too much liquid makes vegetables soggy."
Indexing demon: DEMONO under item: MEAT
by test: Is the item a MEAT?
Indexing demon: DEMONO under item: VEGETABLE
by test: Is the item a VEGETABLE and Is the TEXTURE of item CRISP?

Goal to be activated: Avoid failure of type SIDE-EFFECT:DISABLED-CONDITION:CONCURRENT exemplified by the failure "The broccoli is now soggy" in recipe BEEF-AND-BROCCOLI.

FIGURE 2.3 Repairing Beef and Broccoli:

ANTICIPATOR:
Searching for a plan that satisfies--
Include chicken in the dish.
Include snow peas in the dish.
Make a stir-fry dish.

Collecting and activating tests.
Fired: Is the dish STYLE-STIR-FRY?
Fired: Is the item a MEAT?
Fired: Is the item a VEGETABLE and Is the TEXTURE of item CRISP?

Chicken + Snow Pea + Stir-frying = Failure
"Meat sweats when it is stir-fried."
"Stir-frying in too much liquid makes vegetables soggy."
Reminded of BEEF-AND-BROCCOLI
Fired demon: DEMONO

Based on features found in items: snow peas, chicken, and stir-fry adding goal: Avoid failure of type SIDE-EFFECT:DISABLED-CONDITION:CONCURRENT exemplified by the failure "The broccoli is now soggy" in recipe BEEF-AND-BROCCOLI.

RETRIEVER:
Searching for plan that satisfies--
Include snow pea in dish.
Make a stir-fry dish.
Avoid failure of type SIDE-EFFECT:DISABLED-CONDITION:CONCURRENT

Found recipe -> REC9 BEEF-AND-BROCCOLI

FIGURE 2.4 Anticipating and Avoiding a Previous Failure
In a final note about CHEF, we consider for a moment some of the knowledge CHEF needs to do its job. CHEF's most powerful knowledge source is its case library, which indexes its plans such that they are accessible to guide later planning. Another powerful knowledge source are its TRPs and the strategies they hold that help CHEF repair its failed plans. Another knowledge source is its object critics, which help it to adapt old plans to fit new circumstances. Another powerful but largely hidden knowledge source is its semantic memory, which holds the definitions of the terms it uses. CHEF uses its definitions of terms for a variety of purposes, including partial matching and finding the correspondences between cases. Finding a case that matches best requires both direct matching of symbols and partial matching based on the meaning of symbols. For example, if beef and broccoli is indexed by the goal "include a crisp vegetable," and CHEF is trying to create a recipe for chicken and snow peas, it will know that the beef and broccoli is a good match only if it has defined snow peas as a crisp vegetable. Instantiating an old plan with new objects also requires that CHEF know the definitions of its terms—it needs to know that broccoli and snow peas are both vegetables, for example, to know that one should be substituted for the other. CHEF represents its terms as frames and organizes them with respect to each other in a semantic network.

CHEF proposes a new approach to planning that allows it to avoid many of the problems of more traditional planning systems. Some of its advantages are based on the fact that it reuses plans and therefore can be more efficient than other planners. Some are the result of the fact that it incrementally repairs its plans and learns from the experience. As a result of this learning and the anticipation step performed before attempting to derive a solution, CHEF can predict problems before they happen, allowing it to compose plans that avoid previous mistakes. The combination of these benefits is attained because CHEF views planning as a memory task rather than as a construction task, closing the loop between reasoning, execution, and learning.

### 2.2 CASEY

CASEY (Kotony 1988a, 1988b, 1989) is a case-based diagnostician. As input it takes a description of its new patient, including normal signs and presenting signs and symptoms. Its output is a causal explanation of the disorders the patient has. The causal explanation connects together symptoms and internal states. Figures 21 and 22 in chapter 11 show several of the explanations CASEY has constructed.

CASEY diagnoses patients by applying model-based matching and adaptation heuristics to the cases it has available. It has a case library of approximately twenty-five cases, all of which were diagnosed by the Heart Failure Program (Long et al. 1987). CASEY is built on top of the Heart Failure Program, a model-based diagnostic program that diagnoses heart failures with unprecedented accuracy. CASEY's model-based matching and adaptation heuristics are domain-independent and are as accurate as the domain model they are applied to.

When presented with a new patient, CASEY searches its case library to see if it knows of a similar case and, if so, uses that to diagnose the new patient. If it has no similar case, it passes the case on to the Heart Failure Program, which diagnoses it and returns its results to CASEY to use another time. CASEY uses a two-step process to do case-based diagnosis. First it searches memory for similar cases and uses model-based evidence rules to determine which of the partially matching cases that are retrieved are sufficiently similar to suggest an accurate diagnosis. Then it uses this information to fit the data for the new case into the model.
knowledge CHEF, which indexes useful knowledge shared plans. Another of circumstances, story, which holds variety of purposes, finding a case that is based on the "include a crisp, it will know that a crisp vegetable. definitions of its legs, for example, to mas as frames and

one of the problems is: the fact that it reuses the result of the fact result of this learning, CHEF can prevent previous mistakes, as a memory task, creation, and learning.

it takes a description of symptoms. Its output is a connection together of the explanations adaption heuristics of the five cases, all of seven cases heart failures are relied to.

to see if it knows no similar case, it returns its results to case diagnosis. First to determine which suggest an accurate

<table>
<thead>
<tr>
<th>No.</th>
<th>Feature Name</th>
<th>Value for Old Case (David)</th>
<th>Value in New Case (Newman)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Age</td>
<td>72</td>
<td>65</td>
</tr>
<tr>
<td>2.</td>
<td>Pulse rate</td>
<td>96</td>
<td>90</td>
</tr>
<tr>
<td>3.</td>
<td>Temperature</td>
<td>98.7</td>
<td>98.4</td>
</tr>
<tr>
<td>4.</td>
<td>Orthostatic change</td>
<td>absent</td>
<td>unknown</td>
</tr>
<tr>
<td>5.</td>
<td>Angina</td>
<td>unstable</td>
<td>within-hours &amp; unstable</td>
</tr>
<tr>
<td>6.</td>
<td>Mean arterial pressure</td>
<td>107</td>
<td>99.3</td>
</tr>
<tr>
<td>7.</td>
<td>Syncope</td>
<td>non</td>
<td>on exertion</td>
</tr>
<tr>
<td>8.</td>
<td>Auscultation</td>
<td>murmur of AS</td>
<td>unknown</td>
</tr>
<tr>
<td>9.</td>
<td>Pulse</td>
<td>normal</td>
<td>slow-rise</td>
</tr>
<tr>
<td>10.</td>
<td>EKG</td>
<td>normal sinus &amp; LV strain</td>
<td>normal sinus &amp; LVH</td>
</tr>
<tr>
<td>11.</td>
<td>Calcification</td>
<td>none</td>
<td>mitral &amp; aortic</td>
</tr>
</tbody>
</table>

**Figure 2.5** Differences CASEY Must Reconcile

diagnosis. Then, it applies model-based repair rules (adaptation strategies) to adapt the old diagnosis to fit the new situation.

An example will illustrate. CASEY must diagnose a new patient, Newman. Among other things, Newman is 65 years old, has a pulse rate of 90, a temperature of 98.4, within-hours and unstable angina, mean arterial pressure of 99.3, syncope on exertion, slow-rise pulse, EKG normal sinus and LVH, and mitral and aortic calcification. CASEY's retrieval function returns to it an old patient, David. David was 72, with pulse rate of 96, temperature of 98.7, and so on. Figure 2.5 shows the differences between Newman and David. While there are many similarities between the two cases, there are also many differences, and in order to validate that David provides a good context for diagnosing Newman, those differences must be reconciled.

This is where CASEY's evidence rules come in. These rules specify under what circumstances differences can be reconciled. In general, whenever matching is being done, there will be some features of the old case that match nothing in the new case (e.g., orthostatic change in Figure 2.5) and some in the new case that match nothing in the old case (e.g., syncope and calcification in Figure 2.5). In addition, there will be some dimensions on which some of the values are different in the old and new cases (e.g., age, pulse rate, and temperature in Figure 2.5).

CASEY's evidence heuristics examine the role each descriptor plays in the previous diagnosis and the role all the new descriptors could play in the same diagnosis, and it attempts to match these outlying features to each other. If it is successful, the match is validated. If not, the old case is discarded and another is tried.

For example, one of CASEY's evidence rules says that if two symptoms are both manifestations of the same internal state, then they can be considered to match each other. CASEY's causal model of the heart states that LV hypertrophy can manifest itself as "ekg:lv," or "ekg:lv strain." Applying that rule and using that part of the model, the EKG values in
the two cases can be reconciled with each other. Both play the same role, although their values are different. Another evidence rule says that an extra symptom in the new case that is not present in the old could be a manifestation of some internal state known to exist in the old case. If it is, then the difference is reconciled—the role it plays is as manifestation of a known internal state. This is what allows the aortic valve calcification in the new case to be reconciled with the old diagnosis.

Using its evidence rules, CASEY does, in fact, reconcile all the differences between Newman and David. It goes on to create an explanation of Newman’s disorders by adapting David’s diagnosis. CASEY uses a set of domain-independent model-based repair rules to do its adaptation. Each of its repair rules is associated with one or more of the evidence rules. CASEY keeps track of which evidence rules apply, and it applies the corresponding repair rules at adaptation time. For example, when two values, one of which is present in the new case and one in the old, both play the same role in the diagnosis, a repair rule adapts the old diagnosis to fit the new situation by substituting one value for the other one. When a feature of the new case has been identified as a manifestation of a known internal state, the old diagnosis is adapted by inserting in it an evidence link that connects the internal state to the new symptom. Figures 11.21 and 11.22 in chapter 11 show how CASEY adapted David’s diagnosis to fit Newman. Notice in particular the connections to LV hypertrophy in the lower right corner and the insertion of aortic valve calcification in the upper right corner of Figure 11.22.

Chapter 9 gives the details of CASEY’s evidence rules, and chapter 11 gives the details of its repair strategies. I won’t attempt to describe all of them here. The important thing to remember is that a combination of four knowledge sources work together in CASEY to validate potential matches and to adapt old diagnoses to fit new situations:

1. General-purpose evidence and repair rules predicated on general causal knowledge
2. A causal model of the device being reasoned about (the heart, in this case)
3. The old case, including its problem description and its solution (in the form of a causal explanation)
4. The problem description in the new case

Though the causal model and cases change from domain to domain, the evidence and repair rules are domain-independent and can be used over the wide range of domains for which causal models are available.

One of the major problems that arises in CASEY is that retrieving cases based only on similarity of surface features would result in poor retrieval. In CASEY’s domain, and indeed in many medical domains, the same disorder can manifest itself in many different ways, and different disorders can look very similar to each other at the symptom level. Retrieval based only on available signs and symptoms, then, would cause CASEY to retrieve many inappropriate cases and, even worse, to miss some applicable ones. CASEY compensates for this by indexing cases by both their surface features and the internal states that are part of their diagnoses. Upon retrieval, it infers plausible diagnostic states for its new case before doing retrieval and retrieves based on a combination of plausible diagnostic states and signs and symptoms. A case is considered a good match and a candidate for validation to the extent that its internal
Though their values are case that is not to exist in the old station of a known use to be reconciled.

The differences between orders by adapting repair rules to do the evidence rules. Corresponding repair present in the new rule adapts the old. When a feature of the, the old diagnosis to the new symptom's diagnosis to fit were right corner and e 11.22. 11 gives the details of important thing to in CASEY to validate.

In the usual knowledge is case) the form of a causal inference and repair f domains for which cases based only on domain, and indeed in different ways, and different. Retrieval based only to many inappropriate for this by index-art of their diagnoses. These retrieval and genes and symptoms. To extent that its internal states match the plausible internal states of the new case and, secondary to that, the correspondence between the surface features of the two cases.

Another major concern with respect to CASEY and its domain is accuracy. Can we rely on a case-based program to create solutions to problems requiring accuracy? As stated previously, the Heart Failure Program, which CASEY is built on top of, is an unusually accurate heuristic program with a well-defined causal model guiding its reasoning. Evaluation of CASEY showed that it was equally accurate (Koton 1988a, 1989). If we look at what CASEY added to the Heart Failure Program, we see that it added a set of heuristic rules that are guided by the same model that the Heart Failure Program uses. Our conclusion, then, is that CASEY gets its accuracy from the Heart Failure Program's model, and that its heuristics do not detract from its accuracy.

At the same time, CASEY performs far better than the Heart Failure Program when we measure its efficiency, showing a speedup of two to three orders of magnitude when it holds a relevant case in its case library. Cases provide a starting point for constructing a causal explanation, and the time put into searching the space to construct one from scratch is avoided. Our conclusion, based on these studies, is that case-based reasoning has the potential to speed up our model-based programs considerably without loss of accuracy.

### 2.3 JULIA

JULIA (Hinrichs 1988, 1989, 1992; Hinrichs and Kolodner 1991) is a case-based designer that works in the domain of meal planning. As in other design domains, problems are described in terms of constraints that must be achieved, and solutions describe the structure of an artifact that fulfills as many of those constraints as possible. As in other design domains, problems are large. In general, they cannot be solved by finding one old case that mostly covers the solution and adapting from there. Rather, problems usually must be decomposed into component parts to be solved separately. Yet, component parts interact strongly with each other, requiring some mechanism for keeping track of their relationships.

JULIA solves this problem by combining case-based reasoning with constraint posting and propagation. Its case-based reasoner proposes solutions to problems and warns of the potential for failures. Its constraint mechanisms keep track of the relationships between the component parts of the item being designed. The interaction between the two comes in the indexing vocabulary JULIA uses for its cases. Because constraints describing the relationships between parts of an artifact are so important in synthesizing design components, JULIA uses these constraints to index its designs.

Constraints in design can come from several different places, though all must be treated similarly during reasoning. Some constraints come from our general knowledge of the artifact being designed. For example, in a meal, the major ingredients of dishes should not be repeated across dishes or courses of the meal. Another general rule is that the tastes of dishes in a meal should be compatible. Another is that the meal should be nutritionally balanced. Others are imposed by the new problem. In a particular meal, for example, a client might specify that dishes be easy to make or that some particular ingredients be used or that some dietary constraints be imposed.
In order to create meals that fulfill both general requirements and the requirements of a particular situation, JULIA must use both general knowledge (about meals) and specific requirements imposed by the client as it is reasoning. This means that in addition to using cases as a knowledge source, and in addition to the particular specifications of a new design case, JULIA must have general knowledge about the kinds of artifacts it is designing. JULIA stores this sort of knowledge in its object prototypes. JULIA's object prototypes correspond to different kinds of meals it knows about, for example, normal American dinners, European dinners, buffet-served meals, and one-course meals. Each prototype specifies its structure and the relationships between its parts. A normal American dinner, for example, includes an appetizer course; a salad course; a main course with a main dish and two side dishes, one of which is a vegetable side dish and the other a starch; and a dessert course. They are sequenced in a certain way, and we know, for each, what the approximate size of the dishes are. We also know the relationships between the dishes in the courses. For example, their calorific content, when taken as a whole, should not be excessive, and they should be nutritionally balanced. Their tastes should be compatible, especially the dishes served in the same course. We also know that the more informal the dinner, the more likely that one or more of the courses will be dropped, and several other prototypes (including one-course meals) describe such meals.

JULIA uses both cases and object prototypes as it is doing design. Though it prefers to use cases to suggest its solutions (because they are more detailed), it uses meal prototypes to suggest frameworks for solutions when cases are not available to do that. Though object prototypes are not exactly cases, they play a similar role to that played by cases—they provide large chunks of normative information to the reasoner without the reasoner's having to derive it from scratch.

In a typical problem-solving session, JULIA is told about the new problem. It first attempts to fill in missing information that it knows to be important to meal planning. Some information it is able to fill in by inference; other things it must ask the client. It begins constructing a solution by attempting to find a case that closely matches the specifications of its new problem. If it finds one, it establishes that as its solution framework. If not, it chooses an appropriate prototype and establishes that as its framework. In its next step, it reconciles the requirements of the new situation with the framework it has established. This is where its constraint methods begin their work. The requirements of the new problem are imposed on top of the framework that has been established, and their implications are propagated throughout the framework. In some places there are conflicts. JULIA adapts the framework to fit the new situation in those places. In other places, nothing specific is filled in, but the combination of constraints imposed by the prototype and constraints imposed by the new situation provide guidelines for filling it in. JULIA uses case-based reasoning to fill those in.

JULIA continues by focusing on some aspect of the solution and using case-based reasoning to construct a solution. Each case-based reasoning step involves recalling appropriate pieces of cases that fulfill, as well as possible, the designated constraints for that piece of the solution and then adapting what is retrieved appropriately. Each time JULIA fills in some value in its ongoing solution specification, that value's effects are propagated throughout the rest of the design. When conflicts are encountered, JULIA attempts to adapt the solution to reconcile the conflicts. Failing that, it attempts to adapt the problem specification by mini-

meanwhile relaxing one of these into

meanwhile constructing judging

find a case that

interesting, a

JULIA's case

vegetarians or

such interrupt

attempts to ac

those two met

This so

recounting to
tests had made

make it pour

wanted a brick

had been aim

tion first and

needed to acc

The sec

might recall a

what it is doi

tial for failure

might warn, it

accommodate

exactly as des

JULIA's

11, so I won't substi

The folk

to prepare me

and cheese as

How can

(how)

(gue)

(cos)

(east)

(ing)

JULIA ci

of this problem

bers of jiks-

trying to elabo
2.3 JULIA

mally relaxing some constraints. It resorts to backtracking only if it cannot resolve conflicts by one of these methods.

Meanwhile, there are two interruptions to the process that might happen as JULIA is constructing its solutions. The client might interrupt with a new demand, or the retriever might find a case that warns of the potential for failure. New demands from a client are particularly interesting, as they have the potential to violently disrupt the ongoing design. In one of JULIA’s cases, for example, the client announces late in the design process that there will be vegetarians coming. JULIA must repair its design to accommodate the vegetarians. It treats such interrupts the same way it treats conflicts that arise in the normal course of design. It first attempts to adapt what it has done so far and then attempts to relax constraints, and only after those two methods fail does it resort to backtracking.

This sort of interrupt is common in design, by the way. An architect was recently recounting to me an experience he had designing a major university building. Initially, the clients had made no specifications about the facade of the building. The architects had decided to make it poured concrete. When the design was almost finished, the clients decided they wanted a brick facade. The basic shape of the building was kept, but the structural work, which had been aimed toward the concrete facade, had to be completely redone. Attempting adaptation first and constraint relaxation second before backtracking serves to limit the change needed to accommodate a new specification.

The second type of interrupt JULIA deals with comes from its own retriever, which might recall a case that warns of the potential for failure. When this happens, JULIA stops what it is doing and considers whether the failure might happen. If, indeed, there is the potential for failure in this situation, it updates its specifications accordingly and continues. A case might warn, for example, that there might be vegetarians coming to dinner who need to be accommodated. JULIA asks the client if indeed this is so and, if so, treats it as a new demand, exactly as described above.

JULIA’s adaptation strategies are quite simple and explained in some detail in chapter 11, so I won’t go into detail here. In short, JULIA does two kinds of adaptation. It can make substitutions in its ongoing solution, or it can change the structure of its ongoing solution.

The following sequence shows JULIA’s reasoning as it designs an inexpensive and easy-to-prepare meal for the AI research group at Georgia Tech. It is told that it should use tomatoes and cheese as major ingredients.

How can I help you?
(host tom)
(guests jiks-research-group)
(cost cheap-meal)
(ease-of-prep easy)
(ingredients (tomato cheese))

JULIA chooses its most general meal prototype to start, and propagates the constraints of this problem specification into that framework. For example, it knows that one of the members of jiks-research-group does not eat veal and adds that as a constraint. It continues by trying to elaborate the problem specification through inference and questioning. One inference
it tries to make is the choice of a cuisine. Tomatoes and cheese suggest two cuisines to it, Mexican and Italian, and JULIA asks the client which he prefers, to which he replies, "Italian." Had JULIA been able to decide by itself, it would have.

The decision to serve Italian cuisine requires JULIA to alter the meal framework it has developed so far. In Italian cuisine, there is an extra course, the pasta course, and the appetizer course is specialized to antipasto. It uses its Italian-meal prototype to do this.

JULIA continues by trying to fill in more details about the meal before it actually starts suggesting dishes. It uses recalled cases to do this. Here, it recalls a previous case that was cheap, easy to prepare, and Italian that was informal and served buffet style. It suggests this combination to the client:

What do you think about informal and buffet?  
ok

JULIA goes on to fill in details of the meal itself. JULIA concentrates on filling in details course by course (component by component). However, if it finds a single case that can suggest many of the components, it tries to use that case. Here, the same case that suggested an informal meal served buffet style also suggests a main course: lasagne as the main dish with garlic bread on the side and red wine to drink. JULIA queries the client about those choices.

Would you like lasagne, garlic bread, and red wine?  
yes

This initiates another change to the solution structure, however. Recall that an Italian meal has a standard structure that includes a pasta course. Recall also that meals have general constraints on them that suggest that main ingredients not be duplicated across dishes. The lasagne in the main course violates that constraint. JULIA uses an adaptation critic called "two birds with one stone" to decide that with lasagne in the main course, no pasta course is necessary, and it removes the pasta course from the solution framework. Lasagne in the main dish also conflicts with starch in a side dish, and using the same adaptation critic, it removes the starch side dish from the main course.

JULIA sees that designation of the main course remains incomplete and continues working on it. It focuses now on coming up with a vegetable side dish. It retrieves an Italian dinner with a pasta main dish that had a vegetable side dish and suggests its side dish to the client.

What do you think about brussels sprouts with garlic and parmesan for a vegetable side dish?  
sure. and vegetarian constraint

The client agrees but throws in a new constraint. There will be vegetarians coming to dinner. JULIA propagates this constraint throughout the solution in progress. It further specifies the antipasto in the appetizer course—it must be a vegetarian one. But it conflicts with the lasagne that has already been chosen as the main dish. JULIA attempts to adapt the lasagne to make it suitable. It do the need to ret

There's vegetarian

JULIA is constrained to it uses its adaptetian alternati

How about good

The problem lasagne also sug finished.

JULIA's case same but are fil description, JULIA propagation proc maintaining the is late in the process

1. The goal sc. that provide appetizer. It poses and ac it, but if no goals.
2. The case ret
3. The adaptati violations. It
4. The constrain
5. The reason m
candidate dec
also notices co
6. The structure internally cons

These modules control. The problem:
2.3 JULIA

suitable. It does this by substituting a known variant that is vegetarian. In doing this, it avoids the need to retract everything that depends on the lasagne (here, the garlic bread and the wine).

There’s a contradiction, but we can salvage the plan by substituting vegetarian lasagne for lasagne.

JULIA is finished with the main course and moves on to the appetizer, which is already constrained to be vegetarian and an antipasto. The only antipasto it knows about has meat, but it uses its adaptation rules to delete some ingredients and substitute others to synthesize a vegetarian alternative.

How about vegetarian antipasto for an appetizer?
good

The problem solver now focuses on dessert. The case retrieved previously that suggested lasagne also suggests coffee and spumoni messina for dessert. The client agrees, and JULIA is finished.

JULIA’s case representations are shown in chapter 5. Its problem specifications look the same but are filled in more sparsely. In addition to the processes described in this short description, JULIA uses reason maintenance procedures that work along with its constraint propagation procedures to notice conflicts in its designs in progress. It also has procedures for maintaining the integrity of its designs as it modifies their structures and accepts constraints late in the process. JULIA’s basic structure has six components:

1. The goal scheduler maintains an agenda of design goals. This is the part of the program that provides focus on various parts of the design (e.g., the main dish, a side dish, the appetizer). It also associates plans with its goals that guide the order in which it decomposes and achieves them. JULIA always prefers to achieve a goal without decomposing it, but if no case is available to allow it to do that, the goal scheduler decomposes the goals.

2. The case retriever searches memory for similar cases. Problem descriptors and constraints are used for indexing and ranking cases.

3. The adaptation engine transforms cases and design components based on constraint violations. It uses a small set of primitive transformation rules.

4. The constraint poster (Stefik 1981) propagates values and constraints.

5. The reason maintenance system (Doyle 1979) maintains justifications for and against candidate decisions and records the consequences of decisions and sources of values. It also notices conflicts that are propagated by the constraint poster.

6. The structure maintenance system ensures that the representation of the solution is internally consistent and consistent with the goals of the problem solver.

These modules are coordinated by a control strategy composed of two nested cycles of control. The problem reduction cycle reduces goals to subgoals, schedules them, and evaluates
whether they are achieved. The constraint satisfaction cycle calls upon the constraint poster, case retriever, and adaptation engine to formulate constraints, propose plausible values from cases, evaluate and adapt those values as necessary, choose among alternatives, and propagate constraints.

Helping these processes are some large stores of knowledge. JULIA has several hundred cases. In addition, it knows several hundred dishes and their recipes and knows about the ingredients of each. Its semantic memory holds a taxonomy of concepts such as types of food, dishes, social events, meals, and courses. It also contains problem-independent domain knowledge in the form of plans, constraints, and cases.

Designers talk about three levels of design tasks: routine, innovative, and creative design. Where does JULIA stand with respect to these? When JULIA takes an old design and reuses it, it is doing rather routine, mundane design. Sometimes, however, old designs provide a starting point, but considerable adaptation is needed to merge the suggestions of several old cases or even to make the suggestions made by one case fit the new situation. It is in these circumstances that JULIA moves into innovative and even creative design. I don’t know how we could test where JULIA ranks on the scale of creativity, but discussions with human designers who are considered quite innovative tell us that JULIA is performing many of the same mental acts they perform, using the same sorts of knowledge, and dealing with the same sorts of contingencies and assumptions. Alas, this does not mean that we can see JULIA with knowledge about buildings and ask it to design one. That would be a knowledge engineering feat beyond most of us. But it does suggest that we’ve captured much of what designers do, and that perhaps the model of design that JULIA represents could be the starting point both for creating autonomous design programs and for creating computer programs that can help people do design.

2.4 HYPO

HYPO (Ashley 1990; Ashley and Rissland 1988a) is an interpretive reasoner that works in the domain of law. It was the earliest of the interpretive case-based reasoners, and over its lifetime, it has become one of the most sophisticated. HYPO takes as input a legal situation, and as output it creates an argument for its legal client. It can take the defendant’s or the plaintiff’s side in a dispute and is equally good at creating arguments for either. It’s particular domain of expertise is nondisclosure cases. Typically, in these cases, a secret of some company has been disclosed to a competitor who is then able to take advantage of it, for example, by coming out with a competing product quickly. But not all cases of disclosing trade secrets are illegal, and the task of the program is to decide, for any case, whether or not the disclosure was legal and to create an argument supporting its decision.

HYPO’s reasoning process has several steps in it:

1. Analyze the case for relevant factors. That is, figure out which of its many descriptors are the relevant ones to pay attention to.
2. Retrieve cases that share those factors.
3. Position the retrieved cases with respect to the new one. That is, separate the cases into those that support the point the arguer wants to make and those that support an opposing point.

4. Select the most on-point cases from each set. These are the cases that share the most relevant factors with the new situation.

5. Argue the issue. This is where the arguments get created. In general, HYPO's arguments are what are called three-ply arguments. The most on-point case that can support its point is chosen to make the point. Then the strongest case that makes the opposing point is chosen to counter it. Differences between the two cases are examined, and cases that can address those differences and that support the arguer's point are chosen to rebut the counterargument. Several arguments are made, each based on different initial cases, different counterarguments, and different rebuttals.

6. The analysis done while arguing the issue is used to explain and justify the arguer's point.

7. Hypotheticals are created and used to test the analysis that has been created. Hypotheticals have never happened, but they test the limits of the analysis in ways that real cases might not be able to.

The argument step is the major one in this process. In that step, similarities and differences between old and new cases, and between several old cases, are identified and reasoned about. Chapter 13 holds a concrete example of HYPO's performing each of these steps within the disclosure domain. It is easier, however, to illustrate the process using a simpler, more commonsensical domain.

Here, we imagine George, who is about to turn thirteen, arguing with his overprotective parents about going to the movies to see *Little Shop of Horrors*. They don't want him to go. He wants to go. The first thing George has to do in arguing his case is to analyze the relevant factors. *Little Shop of Horrors* is rated PG-13, and he is not yet thirteen, so age is a factor. Related to age as a factor is maturity. Rating of the movie is also a relevant factor here. Other factors that George knows are important in determining whether one can go to the movies are how far away it is from home, what night of the week it is, and whether one has finished one's homework or not. He determines, however, that none of these is important here.

George recalls cases in which age, maturity, and movie rating were factors. In one, his sister, who is older, was allowed to see the same movie. In another, his friend Noah, who is his best friend (hence of approximately equal age and maturity) was allowed to see it. In another, he was not allowed to see another PG-13 movie a year before. The first two cases support his point; the last doesn't. Of those that support his point, both share the same factors with this one—age, maturity, and rating. Both seem to be equally on point. Thus, George could appropriately create an argument based on either. He decides that the better argument will be based on what his parents have allowed his sister to see; the case of Noah will be used as backup.

1. He creates a three-ply argument based on these cases. First, he argues that he should be able to see the movie because his sister was allowed. He anticipates that his parents will focus

---

1. From (Ashley 1990).
on the differences between the new situation and that of his sister, pointing out that she is older and therefore more mature. He will then use the Noah case to show that neither age nor maturity are necessarily appropriate here, because there is an example of someone of the same age and maturity being allowed to go. The argument itself goes something like this.

George: That's not fair. You let Sarah see that movie. (Sarah is his older sister.)
Parents: Sarah is three years older than you.
George: Why does that make a difference?
Parents: You're not mature enough for that movie.
George: Noah's parents let him go see it. (Noah is George's best friend.)

Of course, George's parents, at this point, might point out that there are factors beyond age and maturity that are crucial. They might point out that the movie is too far away or that George hasn't finished his homework yet. Using these new factors, George might recall other cases and use them to fashion a new argument. Having fashioned that argument, he might test it by creating hypothetical situations. For example, if his parents tell him the movie is too far away, he might recall a case in which a movie was farther away and he was allowed to go. Looking at the differences between those cases, however, he might find that he was allowed to go because another movie his parents wanted to see was playing in the same theater. He might fashion a hypothetical based on that, asking his parents whether, if this movie was playing somewhere where they wanted to go, they would take him there. When a hypothetical case gives positive feedback, it is sometimes possible to shape reality to match the hypothetical. George may be able to show his parents that, indeed, the movie is playing somewhere that is also playing a movie they want to see.

HYPO shows us several important things about case-based argumentation. First, it shows us the importance of compare-and-contrast procedures for understanding situations and makes clear some of the steps involved in that processing and what cognitive processes and knowledge are necessary to engage in such reasoning. Its other contributions are in the specifics it tells us about those steps. For example, it proposes strategies for argumentation. How do we choose the case to use for rebuttal? How will the other side choose a case to use for counterargument? Rebuttal can be simple, but when several cases are available, it is good to use the one that can make the strongest argument. It might make the strongest argument because it covers more points or because it comes from a higher authority (as in the George example), and HYPO proposes strategies for creating hypothetical situations that can help in testing an argument. In addition, HYPO tells us what we, as knowledge engineers, must do to make our programs perform argumentation. In particular, we must seed them with the factors that are the important ones in a domain or give them the capability of computing those factors. The strategies HYPO uses to create arguments, those it uses to create hypothetical cases, and what goes into choosing factors are all presented in more detail in chapter 13.

Though law is HYPO's domain, its areas of expertise are open-ended or open-textured concepts and argumentation, and these are also areas in which it has contributions to make. Chapter 13 has a long explanation of what open-textured means, and I won't repeat that here.

2. Adapted from (Ashley 1990).

2.5 PROTO

PROTO (Bareis 1987; Simmons 1987) is an expert audiologist's emotional case-based reasoning system. It implements both descriptions of situations and knowledge of an item, its knowledge of audiological (hearing patient, PROTO an expert audiologist's emotional case-based reasoning system). Its search to a most likely candidate in the situation or of the closest representation assigned to PROTO's, it repeats the process that is the closest representation to the situation.
ASE-BASED REASONERS

Suffice it to say that many of the concepts that we think we understand well are very poorly defined. For example, a sign on a store might say, "No pets allowed." Does this mean our blind friend with a seeing-eye dog can't take her dog in? A recipe you are following says, "Salt abundantly." How much salt are you supposed to put in? We encounter open-textured concepts on a daily basis, and we must interpret them. We do much of that interpretation based on our experiences. We know how much salt we've used in the past to make pasta, for example, and if the new recipe is one for pasta, we use what we've used in the past—that tells us what "abundantly" means. If we haven't had experience cooking, we are less likely to be able to understand what these ambiguous concepts mean.

Similarly, we all engage in argumentation on a regular basis. Often, we must decide between several alternatives. How do we decide? We weigh the benefits of each. We consider the possible effects. We try to anticipate or project what might happen in each circumstance. We often think about other situations as we are doing this. This process is very similar to the argumentation that HYPO does, and its analysis can give us guidance on building programs that can do commonsense reasoning, on building programs that can help people make decisions, and on the things we need to teach people to make them capable arguers.

2.5 PROTOS

PROTOS (Bareiss 1989a; Porter, Bareiss, and Holte 1990; Bareiss, Porter, and Weir 1988) implements both case-based classification and case-based knowledge acquisition. Given a description of a situation or object, it classifies the situation or object by type. When it misclassifies an item, its expert consultant steps in and informs PROTOS of its mistake and what knowledge it needed to classify the item correctly. PROTOS's domain of expertise is audiological (hearing) disorders. Given a description of the symptoms and test results of some patient, PROTOS determines which hearing disorder that patient has. Its expert consultant is an expert audiologist. It has also been applied to other tasks, for example, recognizing an agent's emotional state.

Case-based classification, as PROTOS implements it, is quite simple. The classification of a situation or object is determined by finding the object or situation it already knows about that is the closest match to the new object. PROTOS assigns to the new item the same classification assigned to its closest match.

PROTOS's searches for a closely matching case using two major steps. First, it narrows its search to a most likely candidate. Then, based on qualities of the match between its most likely candidate and the new item, it follows pointers around the case library in search of better matches. It repeats this second step until it either finds an acceptable match or fails.

This process is a simple implementation of generate-test-debug (Simmons and Davis 1987; Simmons 1988). PROTOS first guesses what category its new problem fits into by looking at how the important features of the new case overlap with important features of categories of hearing disorders it knows about. It verifies that hypothesis by attempting to match its new case to exemplars in the hypothesized category to see if it can find a good match. If it finds a match, it is finished. If it doesn't find a match, it uses the results of its matching process to select a better hypothesis. This process is guided by its knowledge of the kinds of classification mistakes that are common in its domain. In response to misclassifying a case due to strong...
resemblance to cases in the wrong category, PROTOS adds links from the wrong to the right category that allow it to avoid making the same mistake in the future.

A simple example will illustrate. PROTOS must diagnose a hearing ailment described by the following properties:

- s-neural(silk,g2k), ac-reflex-u(normal), ac-reflex-c(normal), o-ac-reflex-u(elevated), o-ac-reflex-c(normal), tym(p)a, speech(normal), air(normal), history(noise), notch-at-4k, static(normal), age(greater-than-65)

It first attempts to narrow its choices. It does this by identifying the categories these symptoms are associated with and choosing the one that is associated with the largest number of important given symptoms.

The strongest reminding is: cochlear-noise

It then attempts to match its new case to the most prototypical of the cases of cochlear noise.

Choosing case p8594R as the most prototypical exemplar of cochlear-noise.
This match is deficient.

PROTOS's strongest reminding is correct about 50 percent of the time. Because PROTOS uses only surface features of its new cases to choose a candidate item for matching, it is not surprising that its first candidate often is not a good match to its new item. PROTOS's next step is to use the results of its match combined with associations built into the case memory to find a better candidate. There are several differences between the prototypical exemplar of cochlear noise and its new case, one of which is that the new case has the property age(greater-than-65) and the old one does not. This particular difference is associated with a link in memory that associates the first case PROTOS considered to another that differs from it by this feature. PROTOS follows that link to the new case and chooses that case as its new candidate case. It considers whether this new candidate case is a good match for its new case.

Traversing difference link labeled age(greater-than-65). Found p827R, a case of cochlear-age-and-noise.
This match is very close.
Patient has cochlear-age-and-noise.

Indeed, the match is a good one, so PROTOS assigns to the new case the classification assigned to the case it matched well: cochlear age and noise.

Two capabilities make PROTOS's case-based classification process work.

1. Its case library records a rich set of semantic connections between its categories and cases allow it to identify likely matches in its case library.
2.5 PROTOS

Its match process can identify correspondences between components of candidate cases and new items even when those components look quite different from each other.

PROTOS has four kinds of connections between its cases and categories. Reminding links associate features with categories, allowing a best guess at a category to be chosen initially. Prototype links connect categories to items that most typify the category. Difference links, or indexes, record important differences between items. They connect items to each other according to their differentiating features. These two kinds of links allow a first best candidate to be chosen from a category, and they also allow movement from candidate item to candidate item, according to the differences between the candidate item and the new item identified by the match procedure. Censor links are also labeled by descriptors and are used to rule out connections that might otherwise be made.

PROTOS's match process uses a combination of two kinds of knowledge.

1. A (often incomplete) model of the item it is attempting to identify that includes functional, causal, correlational, and taxonomic relations

2. Associations between functional components and their implementations in real artifacts or situations

It matches by finding functional correspondences between the components of two items. If it is asked to match a normal four-legged chair to a chair with a pedestal, for example, it uses its model of a chair to recognize that the pedestal on the second chair is playing the same role as the legs on the normal chair, and it allows the match. One match is better than another to the extent that corresponding components are more like each other.

PROTOS's knowledge acquisition process is driven by failures in its classification process. When it is unable to correctly identify the category of an input, it engages in a conversation with an expert that results in the addition of new knowledge and revision of its memory connections. The expert provides PROTOS with appropriate explanatory knowledge that allows PROTOS to associate the specifics of a new case with its functional knowledge. Communication between the expert and PROTOS is through a fixed vocabulary of causal, functional, correlational, and taxonomic relationships. If PROTOS had initially misclassified its case, it also adds difference links to its memory that will keep it from making the same misclassification in the future.

An example will illustrate. PROTOS has just been asked to classify the following disorder:

\[
\text{p827R, a case of}
\]

\[
\text{case the classification}
\]

\[
\text{is work.}
\]

\[
\text{seen its categories and}
\]

\[
\text{air(moderate), bone(normal), speech-intell(normal), tymp-pr(negative),}
\]

\[
\text{tymp-peak(flat), ipsi-AR(absent), contra-AR(absent), other-I-AR(normal),}
\]

\[
\text{other-c-AR(absent)}
\]

PROTOS chooses otitis media as its most likely candidate classification and attempts to match this new case to its most prototypical case of otitis media. All the matches between features of the new case and the prototypical one are direct except for a match between
PROTOS goes on to ask about several correspondences and to review the match that were made. In the new case, the match that were made, the new case is a different case. PROTOS installs it in its matching model. PROTOS has now refined its knowledge about the new case and matched it to a new case. PROTOS installs it in its matching model. PROTOS then installs the new case as an example, so that it can do a better job of matching. PROTOS is now ready to make a decision about the new case. PROTOS decides that it is the appropriate category for the new case.
prototypicality links, while the remaining 10 to 15 percent are through traversal of difference links. PROTOS was 90 percent accurate over its entire 200-case corpus, but after running its entire corpus and engaging in knowledge acquisition discussions with the expert for each case it misclassified, PROTOS was 100 percent accurate in classifying its 24 test cases. After moderate knowledge acquisition, then, PROTOS performs quite accurately, accessing most of its correct matches through prototypicality links but in 15 percent of the cases relying on difference links to do classification correctly.

When we compare PROTOS to other classification programs, the significance of the case-based method of classification becomes more apparent. The most powerful classification program at the time PROTOS was developed was ID3 (Quinlan 1986). When ID3 was seeded with PROTOS's 200 training cases and then tested on its 24 test cases, ID3 was able to correctly classify only 38 percent of the test cases, while PROTOS could correctly classify all of them. In particular, PROTOS excelled on cases with many missing features. In PROTOS's domain, cases are usually described by only about eleven features, while the domain itself is defined in terms of fifty-eight features. Methods like ID3, which define their categories in terms of necessary and sufficient conditions, work best when all examples are described by the presence or absence of all the domain's features. PROTOS's case-based method, by contrast, can work with whatever features are available.

Since then, ID3 has been extended and is a more powerful classifier than it was, able to work with examples with new feature values (Duran 1988). Evidence from PROTOS continues to point to the case-based method's being superior for "outlier" cases, however. Several variants of ID3 that were augmented to deal with missing features and seeded with PROTOS's 200 training cases were still able to correctly classify only 77 percent of the 24 test cases (Porter, Barciess, and Holt 1990). This percentage is roughly equivalent to the 81 percent of the test cases that PROTOS classified correctly through accessing prototypical cases in categories. PROTOS seems to excel at classifying the "outlier" cases, that is, those that are only weakly prototypical whose descriptions share little with the general description of the category itself. Classifying such cases can be confusing, as these cases tend not to fit well into any category—they may have salient features of several different categories (consider whales, for example, or penguins). If PROTOS has seen such outliers in the past and been confused about what category they belong to, it will have installed difference links allowing it to classify such outliers correctly in the future.

2.6 CLAVIER

CLAVIER (Mark 1989; Hennessy and Hinkle 1992; Barletta and Hennessy 1989) is a system for configuring the layout of composite airplane parts for curing in an autoclave. It is up and running at Lockheed in California. Given a set of parts that need curing as input, it designs the

---

3. This number has been disputed by some who claim that the version of ID3 that was used was missing a trivial extension and, as a result, was unable to classify any cases with feature values not present in the training set. For example, if it had never seen a green apple before, this version of ID3 could not have classified a Granny Smith apple correctly. Extending ID3 as it should have been extended would have resulted in some of the misclassified cases being classified correctly. Thus, we probably should not take the specifics of this number seriously. When we discount these cases that ID3 should have been able to classify, however, we still find that PROTOS is more powerful.
layouts for several loads of the autoclave that will cure all the parts, getting as many of them cured on time as possible. As stated earlier, the task of autoclave loading is a black art. A causal model of what kinds of layouts work simply doesn't exist. In fact, the air currents in different autoclaves create different conditions across autoclaves. Thus, experience curing particular kinds of parts in a particular autoclave is necessary before loading can be done efficiently. Fortunately, the expert at Lockheed kept a file of the loads he had done in this autoclave, and Barletta and Hennessy, who built the original system, were able to translate that file into cases. Figure 2.6 shows a schematic of CLAVIER's process.

CLAVIER worked more as it has the one we discuss VIER takes as input with each part. Parts, its cases library well and those that CLAVIER uses the input parts of highest priority.

Figure 2.7 shows the library. The rightmost column holds the second column of several tables, each of which contains CLAVIER's cases. CLAVIER reusing a list of cases in the case library. That is, if two lines are the same, CLAVIER replaces one part could be done simply and choosing one that works.

What CLAVIER does is that it needs to subs in the current table plus so that table suggests.
CLAVIER was seeded with approximately twenty cases and has collected over a hundred more as it has been used. The first version of CLAVIER was fully automated, and that is the one we discuss here, though the later version interacts with workers to do its job. CLAVIER takes as input a list of composite parts needing curing and has a time priority associated with each part. Parts with earlier due dates have priorities higher than those with later due dates. Its case library is made up of layouts of parts in the autoclave, both those that worked well and those that didn't. Cases are indexed by the parts they included. In its first step, CLAVIER uses the input to retrieve cases, choosing as the best case the one that includes the most parts of highest priority.

Figure 2.7 shows sample input and a case from CLAVIER's case library. The leftmost column holds the set of parts that need curing. The middle column shows a case from the case library. The rightmost column shows the solution CLAVIER produces after adaptation. Each case in the case library shows its configuration of parts. Each configuration is composed of several tables, each of which has several parts on it. The long column in Figure 2.8 shows more of CLAVIER's cases.

CLAVIER's most interesting component is its adaptation mechanism. In general, the cases CLAVIER retrieves do not fulfill its needs exactly. Usually, the difference is in one or two parts. That is, in the case that is retrieved, every part in its configuration, except for one or two, is in the list that represents the current situation. CLAVIER's adaptation task, then, is to substitute some part that needs curing for one that is not in its input. One might think that this could be done simply by considering the size or shape of the part that needs to be substituted and choosing one of equivalent size or shape. But for some technical reasons, that doesn't work.

What CLAVIER does instead is to use its experience to guide adaptation. For each piece that it needs to substitute, it looks for a table from a previous load that had the pieces on the current table plus some piece from the input set that still needs curing. It then chooses the part that table suggests. Figure 2.8 shows this adaptation process.
2 / CASE STUDIES OF SEVERAL CASE-BASED REASONERS

The important parts of the process, as one can guess from the illustration, are the context-determination and context-matching steps. Two kinds of knowledge are used for context determination and context matching: global knowledge providing the context in which the table is located and local knowledge describing the table itself. Global context includes such factors as the type of material used by parts in the load, the type of mold being used, and general groupings of parts types (e.g., beams, ribs, stiffeners). It allows the system to find loads with similar heat-up characteristics, thereby allowing the system to find its substitutions among cases with globally similar characteristics. Local context includes mostly spatial knowledge—where in the oven the table is and the sizes of the parts it holds. CLAVIER searches for tables that meet both the global and the local characteristics of the table that needs a part substituted.

On all counts, CLAVIER can be considered a success for case-based reasoning. As an application, it has provided Lockheed with a corporate memory of autoclave configuration knowledge and has been able to apply that knowledge on the fly. As new knowledge is acquired, it is used immediately. The daily experiences of each worker are stored in the system and available to every other worker on the shop floor. This system has already allowed less experienced autoclave operators to load the autoclave like professionals (Hennessy and Hinkle 1992).

CLAVIER was relatively easy to build, partly because the expert on the floor was keeping the right kinds of information in his files. Although loading an autoclave is a black art, the expert who gathered the information understood what it was important to keep track of. The workers on the floor were already using a case-based approach before CLAVIER was built. They kept their records in such a form that they could use them to reconfigure the autoclave later.

As a research vehicle, CLAVIER developed the idea of case-based adaptation and the use of pieces of cases in reasoning, showing the importance of both local and global knowledge in choosing the right pieces of cases. Other systems, too (e.g., CELIA [Redmond 1990a, 1992]), are showing this as a need, and we will discuss it in the chapter on case representation.

The most important advantage reported for CLAVIER has been its ability to learn. It began with about 20 cases, has over 150 at this writing, and continues to grow. As its experience has grown, it has become more accurate in its retrievals, requiring considerably less adaptation now than originally. Its knowledge has grown along with that of the human experts, adapting as they have to changes in the problem space of the domain.

In addition to its successes, CLAVIER demonstrates several difficulties related to fielding case-based applications, particularly in the areas of validating new cases and maintaining the case library. Because no theory of the domain exists, validation of the system requires validation of every individual case. The fact that it learns compounds that problem. CLAVIER is a constantly evolving system. As its case library grows, its behavior changes, making validation even more difficult. Validating individual cases does not ensure that particular cases will be retrieved. Each new case changes the functionality of the system. CLAVIER currently batches its updates to get around this problem, and its creators are working on the problem of continuous validation.
2.7 RETRIEVAL-ONLY AIDING AND ADVISORY SYSTEMS

All the systems mentioned and illustrated so far in this chapter were automated case-based reasoners. That is, given a problem to solve, they did what was necessary to solve it and produced a solution. The only role of a person in those systems was to produce feedback to let the system know how its solution had performed.

But creation of autonomous systems is only one way case-based reasoning can be used for system building. The most powerful thing about case-based reasoning, perhaps, is its fit with what people do. As we stated in chapter 1, people use case-based reasoning naturally in much of their everyday reasoning. In complex domains or those where a person is a novice, however, people do not always remember the most appropriate cases, sometimes because of bias, sometimes because they haven’t yet encountered the appropriate experiences. Many people in the case-based reasoning community believe that case-based reasoning’s biggest potential is in building interactive systems that can help people solve problems or teach them new domains.

The emphasis, as of this writing, in building interactive case-based systems has been on developing interactive aiding and advisory systems. Just as there are several possible roles a human advisor can play in helping us solve problems, so too are there several possible roles a case-based advisory system can potentially play. It can act merely as a browsing facility, it can provide information in response to queries, it can act as a coach, looking over the shoulders of the human and providing guidance and suggestions, or it can act as a colleague, playing a role equal but complementary to that of the human user. Those systems that have been built to date have been browsing facilities and information providers. I begin the discussion, however, with presentation of two hypothetical systems, one a coaching system, the other a query system, to give readers an idea of the range of roles a case-based advisory system can take on.

2.7.1 A Hypothetical Architect’s Assistant

In this hypothetical aiding system, we see the computer acting as a design coach, helping a novice architect with design of a geriatric hospital. There are a number of issues an architect has to deal with: functionality must be appropriate, the design must fit the site, costs must be within limits, and so on. Let us assume that the computer screen is configured with a space for notes, a space for graphical manipulations, a space for the problem specification, and a space where cases are presented. Let us further assume that each case has both a picture part and a textual part. Though the presentation will show all interactions in English, let us also assume an interface in which most of the interaction is through graphics, menus, and pointing, with canned English explanations and descriptions produced by the system.

On the screen we see the new problem in the problem specification space and a representation of the site, showing its contour, size, and shape, in the graphical space.4

**Problem:** Design a geriatric hospital; the site is a four-acre, wooded, sloping square; the hospital will serve 150 inpatients and 50 outpatients daily; offices for 40 doc-

---

4. I thank Craig Zimring for this example.
tors are needed. Both long-term and short-term facilities are needed. It should be more like a home than an institution, and it should allow easy visitation by family members.

Screen: Shows the site, its contour and shape.

The person now uses the mouse to highlight those parts of the problem specification that he or she feels need to be addressed at this point, early in the design.

Person: Highlights hospital, homeliness, provision of visitation, long-term and short-term care, and geriatric.

The computer uses those highlighted parts of the problem as preferences in finding useful cases in memory. The most useful cases will match all those features; less useful ones (that perhaps can address only some part of the problem) will match some subset of those features. The computer finds many cases and partitions them according to how well they match the highlighted criteria. It presents its ordered partitions and asks the person which he or she would like to see more detail on. At the same time, the computer does some bookkeeping (invisible right now). It keeps track of which cases in its lists achieve which of the requested features.

Computer: I know of many long-term facilities for the elderly, many acute-care hospitals, and one geriatric hospital: Wesley Woods in Atlanta. Of the acute-care hospitals, Piedmont Hospital in Atlanta is the most homey. Of the long-term facilities, the Jewish Home in Fairfield County, Conn., is closest in the type of care it provides and the ability to provide opportunities for family visitation. What would you like to see?

The person chooses to see Wesley Woods, the only geriatric hospital with both long-term and short-term facilities. The computer shows a picture of the layout of Wesley Woods and provides a commentary.

Person: Wesley Woods.
Screen: Shows picture and layout of Wesley Woods along with a description window.
Computer: The goal in designing Wesley Woods was to create a homey atmosphere. To do that, they used residential rather than institutional furnishings, forms, and materials and made sure it was cheery and colorful. They made all the corridors short so people would feel as though they were in a home-size place. The functional units of the hospital were hidden from normal sight so the patients didn’t feel they were in a hospital. The design did a very good job of providing homeliness. However, there was one big problem. It was hard for patients and staff to find their way around because the views give no distinguishing landmarks or orienting features and it is not possible to see one’s destination until nearly there. It looked so much like a home that patients and visitors were not sure what to do once they arrived there.
The person makes some notes based on Wesley Woods's successes and failures (make sure corridors have views, make sure corridors have distinguishing features), decides that there are large portions of that design it can borrow, and begins to manipulate that design (using icons) to make it fit the new site. It won't fit on the site because of the slope. The person adapts its design by adding a lower floor, subtracting out part of lower floor of Wesley Woods, putting the two main entries on two different levels.

The computer, which is responsible for bookkeeping, now notices that the building has entries on two levels. It knows this to be a difficult design problem and therefore attempts to retrieve cases that can help the designer. It retrieves several cases that show how to use two entrances. It interrupts the designer and makes a suggestion. The architect makes use of this information.

**Computer:** I have several examples of hospital buildings with entrances on two levels.

Georgia Baptist Hospital has lobbies on two levels, one for inpatients, one for outpatients. Brigham and Women's in Boston has entries on two levels with a two-floor lobby that combines inpatient and outpatient functions. The lobbies are connected by a ceremonial escalator.

**Person:** I'll put the outpatient lobby on the bottom level and the in-patient lobby on the second level.

The computer has kept track of the goals of the designer and how they were achieved in other cases it retrieved that the designer might not have seen yet. It has a mechanism for noticing when design goals are violated by design decisions. In this case, because the inpatient entrance is above usable ground and the Jewish Home in Fairfield had provided for easy visitation by providing easy access to the usable outdoors, the program notices a violation of design goals. It also is able to retrieve another case in which, even though the ground sloped, access to the outside was easily provided. It interrupts the architect to warn of the potential problem and to suggest a solution.

**Computer:** One of the nice features of the Jewish Home in Fairfield was the access to the outside that patients had. That's a potential problem if the inpatient lobby is on the upper floor.

I know of another institution where access was important and was provided by putting patient rooms and gathering places against the same slope so people could walk right outside. In another, the two functions were divided between two different buildings, each on a different part of the slope, so everyone had access to the outside.

We do not continue with the example here, but stop to point out the responsibilities of the machine and the person. The computer has two responsibilities: bookkeeping and retrieval. The bookkeeping it does is of two varieties. First, it keeps track of the good points of designs it remembers. The Jewish Home, for example, had the advantage of providing easy access by patients to the outside. This made visitation by family members pleasant. In addition, it makes
2.7 RETRIEVAL-ONLY AIDING AND ADVISORY SYSTEMS

... easy inferences to make sure, to the best of its knowledge, that good points of designs it knows of are not violated and that bad points are not repeated.

The case retrieval done by the system has three purposes:

1. Cases provide suggestions.
2. Cases provide warnings of potential problems.
3. Cases help in critiquing.

The computer acts as a combination of coach and slave, providing relevant suggestions to the user while acting to serve the needs of the human reasoner. Everything the computer does is in service to the responsibilities of the person using the system. That person is responsible for the decision making itself, doing a variety of reasoning tasks:

- Adaptation
- Choice of features to consider
- Choice of cases to consider
- Evaluation of suggestions
- Evaluation of warnings
- All decision making

2.7.2 A Hypothetical Mediator's Assistant

In the architect's assistant, the hypothetical system looked over the person's shoulder and provided warnings and suggestions as necessary. In this next example, also hypothetical, we see the user requesting cases from the system to help her in answering several questions that need answering to assess the situation and to come up with a solution. This example is based on the reasoning done by PERSUADER (E.P. Sycara 1987a; K. Sycara 1988a, 1988b), a system that acted in the role of mediator in union-management disputes.

In this example, we see the mediator presented with a problem. The union wants a large salary increase, but the company cannot afford it. In order to figure out what to do, the mediator must collect some information. She wants to know why the company is losing money. The system helps her by providing several cases of companies losing money, each of which suggests a different explanation. Note again that we show English interaction because it is easy to show. The person might actually interact with the system in some other way.

**Problem:** Southern Airlines presents its employees with the ultimatum that if they don't take wage cuts of 8 percent, the company, which has become noncompetitive, will have to go bankrupt.

**Mediator:** Get me cases to suggest why the company is losing money although the industry is prosperous.

**Computer:**

**Great Railroad Company:** When they were losing money in prosperous times, it was because the union was insisting on featherbedding, an expensive and unnecessary practice.
Northern Airlines Shuttle: When they were losing money in prosperous times, it was because the management was not treating the employees well. The employees, in turn, were not as productive as they should have been.

Western Trucking: When they were losing money in prosperous times, it was because of mismanagement among the executive board. They were using the resources of Western Trucking for purposes unrelated to the company.

The mediator considers each alternative, collecting information she needs, and decides that this is a case of mismanagement. She now needs to find a solution. She adds to the representation of the problem that there is mismanagement in the company, and asks the computer to suggest solutions. The computer responds with several cases.

Computer:

Western Trucking: Management was forced to resign, and employees took a pay cut until things were back to normal. It took time to get back on the road, but consensus is that the new management did better than the old management could have.

Foobaz Airlines: Employees took a pay cut to avoid bankruptcy but management was forced to put employees on the board and give employees a share in the company. The president took things under control, firing those managers who had been involved in mismanagement. He made it clear that employees on the board were to be taken seriously. Because employees felt both responsible for their share of the company and that they were listened to, productivity went up immediately, and the company began making more money. Two years later the employees not only regained their old salaries but began making more than their counterparts at other airlines. This could not have been done without the willingness of both parties to cooperate in the agreement.

In this hypothetical example, the aiding system is taking a much more passive role than in the previous example. Even so, cases recalled by the system provide the user with much important data. We can think of this system as a smart interface for a database system. It is able to recall partially-matching cases that a decision maker can base analysis and decisions on. It is more than a database in that it does its retrieval based on the specifics of a situation and finds partially-matching cases that can be used to answer the specific questions of the user. It does not require full matching of features, nor does it require a database administrator to formulate queries. Rather, it allows the decision maker to ask the questions himself or herself and to be close to the data decisions are based on.

2.7.3 Some Real Aiding Systems

In the past two years, several real case-based aiding systems have been built and are being developed. The first was the Battle Planner (Goodman 1989). It plays a coaching role for a novice battle planner. The user describes a battle situation to the system and tells the system his or her best guess at a solution. The system then retrieves cases that can be used to critique the solution.
the solution. The Battle Planner also extracts useful summary information from the collection of cases it retrieves. The user uses the cases and summary information to, first, critique his or her solution and, then, if the solution was inadequate, propose a better one. The user again asks the system to retrieve cases and continues in this way until the cases that are retrieved give no more clues about generating a better solution.

A short example will illustrate. Figure 2.9 shows the system's I/O behavior. At the top, we see that the person entered the description of the battle situation and his or her solution. Here, the mission and method fields describe the solution; the other fields describe the battle situation. The user is planning for the American (defender) side. The Battle Planner retrieves nine World War II cases, all in which the attacker wins. It provides commentary about what options and variations were tried in prior cases (and their outcomes) and then supplies a comparative analysis. Armed with that and the cases themselves if the user wants to look at them, the user reformulates his or her solution. This time, the cases retrieved tell the user that the solution is satisfactory.

The Battle Planner is not quite a query system, nor is it quite a coach, but it does perform those pieces of answering queries and coaching that are needed for aiding people doing the particular tasks of projecting the effects of a solution, analyzing those effects, and using those results to reformulate a plan. The Battle Planner has been fielded at West Point with mixed review. Many found it useful; others complained about its idiosyncratic interpretations of battles. It was developed using one of the commercially available shells. The shell it used, REMIND (Cognitive Systems 1992), provides case organization and retrieval capabilities and, based on an influence model given by the system developer, derives summary information useful for critiquing.

Several other simple retrieval-only systems have also been built. CLAVIER, presented earlier, was originally created as an automated system but has been put into production as an aiding system. Users describe a configuration problem, and the system retrieves cases that suggest a solution. The person analyzes whether a case is appropriate, adapts cases as needed, and records any adaptations and how well they worked for future use.

Several help-desk types of systems have been developed or are under development. A help-desk is an information facility set up by an organization to deal with problems users of its systems are having. It is manned by support engineers who provide "over the telephone" service assistance, generally on two kinds of problems: those due to the customer's not knowing how to perform a task and those due to system failures of various kinds. Support engineers are trained to recognize and fix certain kinds of common problems, but they don't generally engage in finding a failure's cause. Thus, a help-desk itself can only deal immediately with the simplest of problems. Harder problems are sent to somebody more expert.

Help-desk implementations, in general, look like smart database systems. The help-desk technician describes a problem to the system, and the system retrieves the closest matching similar cases, which suggest solutions. CASCADE (Simooudis 1991a, b; c: 1992) was created to aid engineers at Digital Equipment Corporation in recovering from VMS device driver failures. The help-desk technician fills in a form describing the current problem, and the system retrieves cases that suggest solutions. Suggestions made by the system allow help-desk personnel to recognize and deal with more of the problems that come their way without passing
### Scenario Situation:
Soviet invasion of Europe, a U.S. Division at Fulda Gap, facing a salient (bulge) in the Soviet line, with a hill behind U.S. troops.

<table>
<thead>
<tr>
<th>Attacker</th>
<th>Defender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nationality</td>
<td>Soviets</td>
</tr>
<tr>
<td>Troop Strength</td>
<td>3700</td>
</tr>
<tr>
<td>Heavy Tanks</td>
<td>54</td>
</tr>
<tr>
<td>Light Tanks</td>
<td>30</td>
</tr>
<tr>
<td>Morale</td>
<td>tired</td>
</tr>
<tr>
<td>Initiative</td>
<td></td>
</tr>
<tr>
<td>Terrain</td>
<td>Rugged, mixed</td>
</tr>
<tr>
<td>Mission</td>
<td>Seize hill</td>
</tr>
<tr>
<td>Method</td>
<td>Frontal assault</td>
</tr>
</tbody>
</table>

**Retrieved Cases:** 9 cases from WWII, all attacker wins.

- In one battle, rapid assault → major victory
- In two other battles, delaying actions → successful second defense

**Comparative Analysis:** Significant factors generated by retriever from its clustering:

These factors favor Attacker win:
- Defender lacks reserves
- Defender lacks depth

**New Mission and Method:**

<table>
<thead>
<tr>
<th>Attacker</th>
<th>Defender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nationality</td>
<td>Soviets</td>
</tr>
<tr>
<td>Mission</td>
<td>Seize hill</td>
</tr>
<tr>
<td>Method</td>
<td>Frontal assault</td>
</tr>
</tbody>
</table>

**Retrieved Cases:** 18 cases, all defender wins.

**FIGURE 2.9** An Example from the Battle Planner

...
that, they built an organizational memory into their system and integrated the case-based support system with other help facilities that enable users to get short descriptions of tasks they need to do and identify others in the organization with whom they might collaborate in solving problems.

More recently, developers have become more ambitious, attempting to build systems that aid more complex problem solving. SCI-ED (Kolodner 1991b), for example, will help elementary school teachers plan science lessons. ARCHIE-2 (Domeshek and Kolodner 1991, 1992, 1993) is designed to help architects with conceptual design of buildings. AskJef (Barber et al. 1992) is being developed to help computer interface designers make appropriate choices in laying out interface screens. The ASK systems (Ferguson et al. 1992) are browsing tools that have education as their goal. ASK-Tom teaches novice bank consultants their job. Adviser, the President teaches high-school students about contemporary American history. Common to all the systems just mentioned is the use of a kind of case called a story. A story is a way of presenting a case so that the lesson it teaches is clear. The same case might be presented through several different stories, each of which provides a different point of view or focuses on different aspects of the case.

In story-based systems, the premise is that the user will browse through a series of stories in each sitting, stories that are related to each other through some natural progression of goals the user has. An architect, for example, may start with the goal of investigating acoustics issues, concentrating first, say, on a particular type of room (e.g., a courtroom), moving on to another type of room, moving from there to heating and air conditioning in the same room, and so on. A novice learning about bank consulting may start by focusing on the job itself, move on to finding out more about a particular aspect of the job, look at alternatives for doing some particular task, and so on. The important thing in these systems is to make sure the system allows these natural progressions. Thus, a big issue in creating these systems is to discover a presentation scheme that clusters related stories together.

The ASK systems address this issue by using the metaphor of a novice having an advisory conversation with an expert and having interactions with the system on conversational moves that make sense in context (Schank 1977). Given some item the novice has just asked about, the screen makes available follow-up questions the novice would be likely to ask next. An illustration from ASK-Tom (Schank et al. 1991; Ferguson et al. 1992), shown in Figure 2.10, will make this more concrete. In the middle of the screen, we see the issue the user is currently interested in—first things to find out about customers and their assets. The user, we assume, has just heard a story giving advice about that issue. Related issues the system knows about are organized around the circumference of the screen. Related stories might provide context for the advice just given, give additional detail, and so on. ASK-Tom groups these stories along several dimensions natural to a continuing conversation with the expert. Thus, the user can continue by asking for alternatives to doing what was proposed in the focal story, examples that show more specific issues that arise, expected results of following the advice of the story, warnings about what to watch out for in carrying out the advice, opportunities that might be taken to better carry out the advice, and context that might be important to the advice. Pointing and clicking on one of those boxes enables the user both to hear the story that goes with the box and to consider other relevant issues related to the new focus.
FIGURE 2.11 A Screen from ARCHIE-2

Other systems organize their stories in a graphical presentation scheme. ARCHIE-2, which gives advice about architecture, clusters its stories using a floor plan, a representational form architects are used to reading. The floor plan is annotated with stories when there is something significant to get across to the designer. Some stories in ARCHIE-2 tell about design features that didn’t work and what could be done to fix them; others report on features of the design that were particularly, and perhaps unexpectedly, good. Users first describe the system they are working on, and the system retrieves buildings that are similar to the new one. The user then asks to see some aspect of the design of a building that is retrieved, and is shown a floor plan surrounded by annotations. Figure 2.11 shows an example. At the top of the figure is a floor plan with annotations around it, each giving a short description of some design feature. Beside the floor plan are controls and forms that allow users to specify design issues and floor plan pages they are interested in. At the bottom of the figure, we see what results when the user clicks on one of those annotations: additional detail about the designated story is provided, along with buttons that allow access to guidelines and other examples of similar phenomenon. AskJef, which gives advice about layout of computer interfaces, uses the layout of a computer screen similarly to organize its stories. This means of clustering seems to make sense when the advice being given is about an artifact and how it works or is designed.
The most ambitious of these story-based browsing systems is the Story Archive (Bareiss, Ferguson, and Fano 1991). It holds a wide range of stories, in the form of video clips, covering historical events, entertainment, famous people, and so on. They are indexed and cross-indexed according to the issues and topics each addresses. Using an interface like the one from ASK-Tom, a user can ask for a video clip about some topic or issue and from there explore video clips about related topics. The system allows users to explore issues from several different points of view to find out how different political groups, religious groups, or ad hoc groups have analyzed different issues. The idea is to create an environment in which a person can learn both facts and analytical thinking skills in an enjoyable way.

Indeed, it is not a far leap from browsing systems that advise to those that teach, though some creative thinking and interface design is needed to make that leap. Advise-the-President, for example, began as a browsing system in which a student could find out about decisions presidents have made (e.g., about the Iran hostage crisis). Its most recent incarnation sets the student up as an advisor to the president, asking the student to find out as much as he or she can about an issue and the way it has been approached in the past in order to give the president advice. The indexing, presentation of stories, and connections between stories remained the same when the system moved from being a browsing system to an instructional system anchored by a task the student had to do. Changes were in the interface only.

The creation of case-based aiding systems has begun in earnest only recently, and it is too early to say exactly what their contribution will be. The potential benefits of such systems are quite exciting to think about, however (Kolodner 1991a). For novice problem solvers, such systems can provide the range of experiences they haven't had, allowing them to solve problems based on the wisdom of experts. There are several reasons novices should be able to perform better using such aiding systems than without them. First, with more cases available, they will be able to recognize more situations and the solutions or evaluations that go with those. Second, if cases that are available include failed cases, novices will be able to benefit from the failures of others. Third, novices will have available to them the unanticipated successes, and therefore the tricks, of experts that they wouldn't have otherwise. Fourth, retrieved cases will allow novices to better recognize what is important in a new situation. Cases indexed by experts and retrieved on the basis of a description of a new situation will be those that experts would recall and will show the novice ways of looking at a problem that he or she might not have the expertise for without the system. Fifth, the ability to recognize what is important will allow for better critiquing of solutions and situations. Additionally, novices will have access to obscure cases that they otherwise would not be able to make use of. These obscure cases can help with any of the tasks listed above.

Using these systems during a training period additionally provides students with a model of the way decision making ought to be done, for example, what things ought to be taken into account, and provides them with concrete examples on which to hang their more abstract knowledge. Much of the expert decision-making skill people have comes from observing experts and discussing with experts why they solved problems in certain ways. A case-based aiding system can provide at least some of that experience.

Benefits of these systems are not just for novice problem solvers, however. In some domains, there is much to remember. For tasks where there is much to remember, case-based aiding systems can augment the memories of even expert decision makers. In addition, as dis-
2.8 SUMMARY

Six automated case-based reasoners were presented in this chapter. CHEF is a case-based planner. JULIA is a case-based designer. CASEY is a case-based diagnostic program that creates causal explanations using a combination of case-based reasoning and model-based knowledge. HYPO does interpretive reasoning in the legal domain. PROTOS does case-based classification and its case-based procedure is far more accurate than inductive approaches. CLAVIER is used at Lockheed to lay out composite parts of airplanes in an autoclave, and has been a major success as an application. It also implements a novel case-based adaptation strategy.

While this chapter has concentrated on automated case-based systems, the next chapter concentrates on the cognitive model implied by case-based reasoning. It first presents the model and then continues by discussing the implications of the cognitive model. One implication, if we consider case-based reasoning to be a good model of people, is that it can serve as the basis for guiding the creation of interactive systems that can advise and aid people doing complex cognitive tasks. Several such systems that have already been built will be discussed in that chapter.