

HaleyAuthority Tutorial

Revision 1.1 – 8/18/07



Contents

1	Introduction to HaleyAuthority	4
1.1	Scenario	4
1.2	Computers that hear and obey	6
1.3	Advantages of HaleyAuthority	7
1.3.1	Advantage of rules	7
1.3.2	Advantage of natural language.....	9
1.4	Haley Systems, Inc.	10
2	Managing policy with HaleyAuthority.....	11
2.1	Scenario	11
2.2	Grammar and linguistics.....	12
2.2.1	Parts of speech	12
2.2.2	Grammatical roles in a sentence	13
2.3	Business concepts	14
2.3.1	Entities	15
2.3.2	Values	15
2.3.3	Relations	16
2.3.4	Phrasings	18
2.4	Statements and rules	19
2.5	Scenario	21
3	Writing policies clearly	22
3.1	Scenario	22
3.2	Clarifying the statements.....	23
3.2.1	Find the unstated assumptions.....	23
3.2.2	Identify the concepts	23
3.2.3	Minimize negative logic	24
3.2.4	Investigate limits.....	24
3.2.5	Split <i>or</i> into multiple statements.....	25
3.2.6	Minimize punctuation.....	25
3.2.7	Minimize plurals.....	26
3.2.8	Avoid mass nouns	26
3.2.9	Avoid pronouns	27
3.2.10	Using <i>a</i> , <i>an</i> , and <i>the</i>	27
3.2.11	Use present tense	29
3.2.12	Minimize modal verbs.....	29
3.2.13	Minimize <i>then</i>	29
3.3	Scenario	30

4	Educating HaleyAuthority	33
4.1	Scenario	33
4.2	Creating a new knowledgebase	33
4.3	Modules, statements, and applicability conditions.....	35
4.3.1	Adding a module	35
4.3.2	Adding a statement	36
4.3.3	Adding an entity and a noun phrase.....	37
4.3.4	Adding a verb phrase	40
4.3.5	Adding an applicability condition	47
4.3.6	Adding the <i>person</i> and <i>smoker</i> entities	47
4.4	A person's age	53
4.4.1	Adding a value	56
4.4.2	<i>theAgeOfAPerson</i>	57
4.4.3	Creating a point-and-click statement.....	59
4.5	Hazardous occupations.....	60
4.5.1	Adding the entity <i>occupation</i>	60
4.5.2	Defining the relation an occupation is hazardous.....	61
4.5.3	Adding instances of an <i>occupation</i>	63
4.5.4	Adding a fact to an <i>occupation</i> instance.....	64
4.5.5	Adding the phrasing <i>an occupation of a person</i>	65
4.6	The applicant's coverage and income	66
4.6.1	Adding the relation <i>an application requests coverage for an amount of money</i>	66
4.6.2	Adding the phrasing <i>an income of a person</i>	70
4.7	The applicant's answers.....	70
4.7.1	Adding the entities <i>question</i> , <i>health question</i> , and <i>answer</i>	71
4.7.2	Adding instances of a <i>health question</i> and instances of an <i>answer</i>	71
4.7.3	Adding the relation <i>aQuestionOnAnApplicationWasAnsweredWithAnAnswer</i> ..	72
4.8	Height and weight limits	75
4.8.1	Adding the concepts <i>body mass index</i> , <i>square</i> , <i>height</i> , and <i>weight</i>	77
4.8.1.1	<i>Body mass index</i> and <i>square</i>	77
4.8.1.2	<i>Height</i> and <i>weight</i>	78
4.8.2	Adding the relations <i>theHeightOfAPerson</i> , <i>theWeightOfAPerson</i> , <i>theBodyMassIndexOfAPerson</i> , and <i>theSquareOfANumber</i>	80
4.8.2.1	<i>theHeightOfAPerson</i> , <i>theWeightOfAPerson</i> , and <i>theBodyMassIndexOfAPerson</i>	80
4.8.2.2	<i>theSquareOfANumber</i>	81
4.9	Adding the Calculation Module.....	82
4.10	Final disposition rules.....	83
4.11	Scenario	86
5	Applying policy to test cases.....	88
5.1	Scenario.....	88

5.2	Applicant instances	88
5.2.1	John Doe	88
5.2.2	Jane Doe	92
5.3	Running a test case	93
5.3.1	Defining the test case	93
5.3.2	Trace messages	95
5.3.3	Deploying the logic for a test case	95
5.3.4	Results of the test case	96
5.4	Scenario	99
5.5	HaleyAuthority and your business	100

1 Introduction to HaleyAuthority

Welcome to the *HaleyAuthority Tutorial* from Haley Systems, Inc.. This manual shows you how to implement a small application-approval system using HaleyAuthority. The example concerns the automated approval of life insurance applications from a Web site. The early chapters explain concepts and best practices. The later chapters show how to navigate the user interface to teach HaleyAuthority about your business. The final chapter demonstrates how to run the resulting program and validate that it performed as intended.

In this chapter, we introduce HaleyAuthority and a typical new-user scenario. We will follow this scenario chapter-by-chapter as it unfolds.

1.1 Scenario

Michael Marks sat in the rear of the Friday morning meeting, hoping to be inconspicuous. He was the new person in the Information Technology department, and was still learning the jargon of the life insurance business. The head of IT was almost finished with his PowerPoint proposal for the Web site update. The company CEO and his two vice presidents didn't look very happy.

"Are you saying," growled the CEO, "that this revision to the Automated Approval Application is going to take half a year to implement?"

“At least,” replied the IT manager, shutting off the projector. “It will take four months to write the code, and several weeks more to test and validate.” He sat down. “If you come up with any new policies in the meantime, it will take even longer.”

“Why so much time?” demanded the CEO. “We need those new policies on line now. We can’t stay competitive if our approval policies are months behind the market.”

“The AA Application is a very intricate business program,” replied IT. “It is an upgrade to our original application triage program, which was written in Java and VBScript, and the new policies have to be integrated with the existing ones. Adding new policies to a decision tree is very error prone. It takes time.”

The CEO looked disgusted. “How long would it take to change just one guideline?” he asked. “Suppose we need to change the minimum age of a male applicant? We need to raise the minimum age from 18 years to 21 years, for men only. How long to implement a change like that?”

“For men only?” The IT manager glanced around the room, getting estimates from his engineers. “A week.” “Ten days.” “No, something would go wrong. Call it two weeks.” “Don’t forget testing.”

“A month,” IT replied.

The CEO was incredulous. “A month? To change one guideline? How can it take that long?” he demanded.

“Well, the code is pretty complex, and we’d have to find all the branch points where we test for age and introduce new branches dependent on gender. I’m not sure if the code checks for gender now. We would have to trace through the code to be sure that those changes didn’t interfere with other policies. We’d have to run regression checks and track down any change in behavior.” He spread his hands helplessly. “It is a non-trivial task.”

“Wait a minute,” said the CEO. “Why don’t we know if the code checks gender? Don’t we know what this application does?”

“The prototype system was coded by a graduate student last summer,” explained an engineer. “He went back to school. He didn’t comment the code.”

“You’re telling me that our automated approval software, which approves hundreds of applications every day, is so complicated that we don’t really know what it does?” The CEO had a dangerous look in his eye.

No one replied. The room was very quiet.

“And it would take weeks to change any part of it?” More silence.

The CEO became very serious. "The Board of Directors wants us to be up to date. They want us reacting to the changing marketplace. They want us to be competitive, to adapt quickly to changing business conditions." He glared at the engineers. "There will be a board meeting Monday. They will ask me how the AA Application is coming, and this isn't the story I want to tell them."

The developers looked like a herd of deer caught in the headlights of a car.

The CEO slumped back in his chair. "Why can't we just tell computers what to do in plain English and have them do it?"

Mike started to speak and then caught himself. Too late. The CEO's eyes snapped around in Mike's direction. "You're the new guy, right? You have an idea?"

Mike swallowed. "Well, sir, I met a guy on a plane flight last week. He was very excited about a business system you could program in plain English." The CEO frowned but Mike hurried on. "He just wouldn't let go of it, and finally insisted on giving me a demo disk. I still have the disk in my briefcase."

"And?" inquired the CEO. Behind him, the head of IT was motioning to Mike to be quiet.

But Mike was trapped. "Well, I suppose it wouldn't hurt to install it and try it out."

The CEO never dithered. "Good. Try it this afternoon. Get some approval guidelines from underwriting and see what you can do."

1.2 Computers that hear and obey

Computers are appearing in every aspect of our daily lives. They organize our business information, transmit our messages, modulate our music, diagnose our illnesses, predict our weather, remind us of our anniversaries, and route our telephone calls to us. There are computers in our offices, homes, automobiles, wristwatches, kitchen appliances and even inside some of our credit cards. There may be a computer in your hip pocket at this very moment.

How many of these computers *understand* us when we speak to them? Voice-recognition has come a long way in the last few years. Anyone can purchase inexpensive voice-recognition software that interprets spoken words into editable text with 95% accuracy. You can say "Call home!" to your cellular phone, and it will usually dial *your* home rather than someone else's. Simple voice recognition is a reality. Suppose you were to turn to your cellular phone and say, "Hold my calls for the next thirty minutes, unless it's my doctor." How would your phone respond? Would it hold your calls? Would it screen them as directed? No?

It isn't a question of recognizing the words. The problem is the same when you type a sentence directly into a field on a form. The computer "knows" exactly what you *typed*. It does not know what you *meant*. It doesn't *understand*.

The goal of Haley Systems, Inc. is to bridge this gap. Computers should *hear and obey*, like the genie in Aladdin's lamp. Aladdin conversed directly with the genie to make his three wishes. The genie obeyed. If the genie had acted like an e-commerce Web server, he would have dropped thirty pounds of software manuals in Aladdin's lap and told him to spend a year coding his first wish.

In the next few years it will become commonplace to speak directly to everyday devices and appliances. We will be able to call them on the phone or send them email, and they will respond intelligently to our instructions. We will ask the phone if we missed any messages. We will ask our laptop the names of the prospects we visited yesterday. It will answer. We will tell the laptop to send the prospects the usual follow-up letters. The laptop will send the letters.

To make this possible, computers and humans will need to speak a common language. The first such language will be English. HaleyAuthority is the application that proves this dream is attainable. HaleyAuthority understands. It obeys.

Not "someday." *Today*.

1.3 Advantages of HaleyAuthority

HaleyAuthority is the natural-language front-end to Eclipse, which itself is a product of 20 years of experience in commercial rule applications. Combining natural English interpretation with rule-based systems gives HaleyAuthority all of the advantages of rules without the cost of programming them. *HaleyAuthority programs itself*.

1.3.1 Advantage of rules

Modern business systems implement policies. An insurance or warranty system may have to apply hundreds, even thousands, of individual guidelines to an endless stream of incoming data. Traditional procedural languages, including those used in Web applications, cannot meet this challenge in a flexible and timely manner.

The problem is flow of control. A traditional computer program takes the flow of control down a single path, branching where needed, until it reaches a block of code that performs some desired action. This approach is fine when the code has only a few branches, but it becomes unmanageable if the system must make many branching decisions.

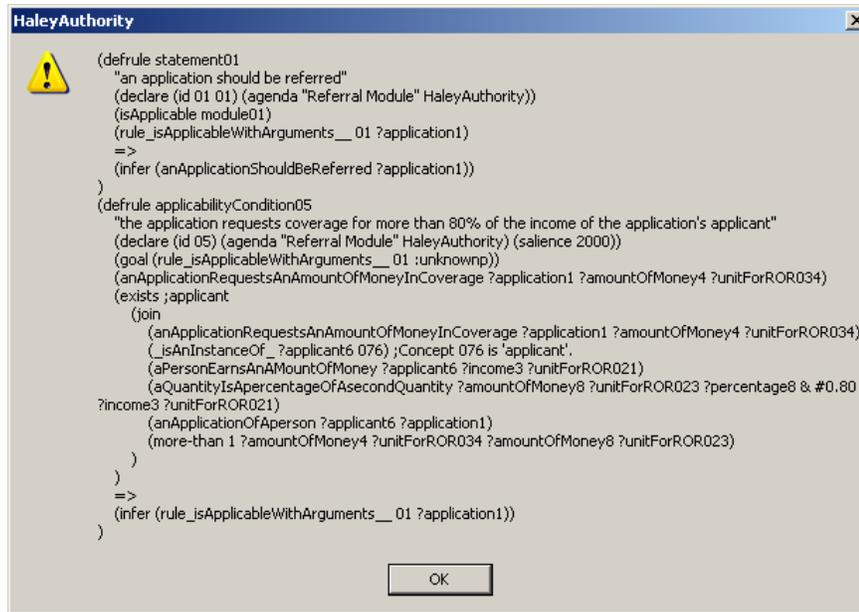
For instance, suppose you have policies that apply ten independent yes/no decisions, and then take some action. The developer implements the first decision and the code takes two branches. If the second decision is independent of the first one, the programmer may implement it twice (once on each branch) and suddenly there are four branches. By the time the programmer gets to the tenth decision, the decision tree may have over a thousand branches.

Admittedly, this example is unrealistic. System analysts make a living paring down decision structures and combining branches to reduce needless repetition. They build clever mechanisms to route the flow of control efficiently down the minimum number of necessary paths. The resulting structure is more elegant and efficient, but it is difficult to revise. Asking for a trivially small change to a decision system can be like asking your mechanic to add ten more horsepower to your car's 200-horsepower engine. That sounds like a small change, but it requires rebuilding the whole engine.

There are often thousands of decisions to implement. Automobile manufacturers support immense decision systems to approve or deny payment for warranty repairs. They receive thousands of such requests every day from mechanics and dealers. Does the warranty cover this part for this service on this model sold on this date in this state? Is the warranty still in force? Is there any sign of fraud? Over the years, the number of potential tests and decisions can become overwhelming. This is where rules come in.

A rule system avoids the flow-of-control tangle by using small, independent statements of tests and actions (the rules). Because there is no overall flow of control, there is no decision structure to maintain. Each policy is implemented as a single, separate, independent rule. To add a new policy to the system, you write a single new rule. To remove an old policy, you delete a rule. You can change a policy freely without fear of damaging the rest of the program.

For instance, we might have a policy that detects when a life insurance applicant has asked for too much coverage: "An application should be referred if the requested coverage is more than 80% of the income." As the subject-matter expert, you would just type this policy directly into HaleyAuthority. The illustration below shows how HaleyAuthority takes your statement and implements it as a rule in Eclipse.



```

(defrule statement01
  "an application should be referred"
  (declare (id 01 01) (agenda "Referral Module" HaleyAuthority))
  (isApplicable module01)
  (rule_isApplicableWithArguments__ 01 ?application1)
  =>
  (infer (anApplicationShouldBeReferred ?application1))
)

(defrule applicabilityCondition05
  "the application requests coverage for more than 80% of the income of the application's applicant"
  (declare (id 05) (agenda "Referral Module" HaleyAuthority) (salience 2000))
  (goal (rule_isApplicableWithArguments__ 01 :unknownp))
  (anApplicationRequestsAnAmountOfMoneyInCoverage ?application1 ?amountOfMoney4 ?unitForROR034)
  (exists ; applicant
    (join
      (anApplicationRequestsAnAmountOfMoneyInCoverage ?application1 ?amountOfMoney4 ?unitForROR034)
      (_isAnInstanceOf_ ?applicant6 076) ;Concept 076 is 'applicant'.
      (aPersonEarnsAnAMountOfMoney ?applicant6 ?income3 ?unitForROR021)
      (aQuantityIsAPercentageOfAsecondQuantity ?amountOfMoney8 ?unitForROR023 ?percentage8 & #0.80
        ?income3 ?unitForROR021)
      (anApplicationOfPerson ?applicant6 ?application1)
      (more-than 1 ?amountOfMoney4 ?unitForROR034 ?amountOfMoney8 ?unitForROR023)
    )
  )
  =>
  (infer (rule_isApplicableWithArguments__ 01 ?application1))
)

```

Remember that you just type in the policy. HaleyAuthority creates the rest of the rule for you. With HaleyAuthority, no one on your staff has to write this kind of code.

The conditions of the rule are listed above the inference arrow (\Rightarrow), and the consequent of the rule is below the arrow.

These few lines of code perform a *real policy function*, and can be added, modified or removed without any effect on the rest of the system. You can change *any* policy without redesigning the whole program. Suppose 200 actual horses, instead of a 200-horsepower engine, pulled your car. Adding 10 more horses to the team would not be difficult. *You could make that change in a few minutes*. Rules make that difference in a decision system.

1.3.2 Advantage of natural language

We showed you that Eclipse rule to make an important point. The Eclipse code looks quite complicated. No problem. With HaleyAuthority, *you never have to write this kind of code*. You write a *statement*, which is a simple description of the rule in plain English. HaleyAuthority creates the Eclipse code for you behind the scenes. It programs itself. For instance, HaleyAuthority created that Eclipse rule from this statement and applicability condition: Anyone can read this:

S an application should be referred
 **A** Applicability
C if: the application requests coverage for more than 80% of the income of the application's applicant

Learning to use HaleyAuthority involves a short initial investment of time on your part. The *HaleyAuthority Tutorial* presents all of the necessary concepts and skills. Simply

follow along as Mike Marks builds his first HaleyAuthority prototype. Mike encounters every situation that puzzles new users, and we have made sure that he finds the right answers.

It takes a couple of hours to work through the example. It is time well spent, and you may even enjoy it.

1.4 Haley Systems, Inc.

Haley Systems, Inc. has been the recognized global leader in rule-based programming, as well as a leading expert in automating managed knowledge, since 1989. Founder Paul Haley has been a leading figure in the commercialization of artificial intelligence for more than 20 years.

Haley Systems, Inc. empowers business people to express, access, and maintain their business knowledge, best practices, and policies in plain English sentences. Haley System's artificial intelligence (AI) embeds these sentences within and integrates them with other information technology (IT) to automate their knowledge – without a software development cycle. Solutions from Haley Systems, Inc. operate on the full range of platforms, from mainframes to hand-helds.

Many of the world's largest companies embed software from Haley Systems, Inc. in a variety of commercial software packages and applications. Haley System's solutions are used broadly in customer relationship management throughout the financial services and telecommunications sectors, as well as other industries. Web applications using Haley Systems' multi-threaded server software execute business rules several times faster than any alternative and with no discernable latency.

2 Managing policy with HaleyAuthority

HaleyAuthority provides a natural-language interface to Eclipse, the enterprise-class rule engine from Haley Systems. HaleyAuthority lets your managers implement company policy in a completely natural way, using simple English sentences. The policy makers do not need to understand any of the underlying technology in order to exercise full and immediate control over the system. A policy maker can literally edit a sentence, save it, and see the system's behavior change.

Before you reach that point, however, someone will have to prepare HaleyAuthority for management's policy input. We will call this person the *knowledgebase administrator*. The administrator determines the general architecture of the system, and educates HaleyAuthority about the words and phrases used in the local business environment. The administrator does not need to understand HaleyAuthority's underlying technology in detail, but it is always helpful to have a general grasp of how the system works.

This chapter will give you some background about how HaleyAuthority operates. If you have not worked with a rule-based system before, this chapter will help demystify the technology. The material is not difficult, but it is *different*.

2.1 Scenario

Mike Marks returned to his office and located the HaleyAuthority demo disk. He called Haley Systems for a temporary authorization code and opened the application. As he expanded the nodes in the knowledge tree, he recalled Miss Higgenbottom's sixth-grade grammar class. Mike had been the kid who was good at science and math. He dreaded the thought of revisiting basic grammar.

HaleyAuthority's Dictionary folder contained nodes for nouns, verbs, adjectives, adverbs, determiners, and prepositions. Mike remembered nouns, adjectives and verbs. They were easy. He thought he remembered what prepositions were, but adverbs? They ended in "ly," didn't they?

As he continued to explore HaleyAuthority, Mike made a list of other items he wasn't sure about. Entities? Relations? Phrasings? He tried to edit a phrasing and encountered

options for modals and auxiliaries. Either Miss Higgenbottom hadn't mentioned those, or Mike had been dozing when she did.

Mike spent a few minutes searching the Web for English grammar sites. He reacquainted himself with nouns, adjectives, verbs, adverbs, determiners, prepositions and the other basic parts of speech, but quickly discovered that relations, entities and phrasings were not listed. He sighed in frustration and grimly opened the online help; Mike hated to read documentation.

2.2 Grammar and linguistics

HaleyAuthority embodies a very sophisticated model of linguistic relationships. In fact, HaleyAuthority knows quite a bit more about the structure of language than Miss Higgenbottom did. This makes HaleyAuthority's interpretation of English very powerful (which means less work for the knowledgebase administrator).

HaleyAuthority cannot write a poem or a play, but it can follow directions written in clear, unambiguous English. Before you can educate HaleyAuthority about the objects, processes and jargon of a specific business, you need to refresh your memory of the basic parts of speech and the roles they play in the construction of a typical sentence. This section is a brief review of these concepts.

It seems that quite a few of us were dozing in class that day.

2.2.1 Parts of speech

Miss Higgenbottom may have learned the parts of speech as a little girl by memorizing this Victorian rhyme:

Three little words we often see,
Determiners like *a*, *an*, and *the*.

A *noun's* the name of *anything*,
A *school* or *garden*, *hoop*, or *string*.

An *adjective* tells the kind of noun,
Like *great*, *small*, *pretty*, *white*, or *brown*.

In place of nouns the *pronouns* stand,
John's head, *his* face, *my* arm, *your* hand.

Verbs tell of something being done,
To *read*, *write*, *count*, *sing*, *jump*, or *run*.

How things are done the *adverbs* tell,

Like *slowly, quickly, ill, or well.*

A *preposition* stands before

A noun, as *in* a room or *through* a door.

Conjunctions join the nouns together,

Like boy *or* girl, wind *and* weather.

The *interjection* shows surprise,

Like *Oh, how charming!* *Ah, how wise!*

The whole are called nine parts of speech,

Which reading, writing, speaking teach.¹

A Victorian child could earn a grade of 99% by reciting this poem correctly. The student could then earn the final 1% of the grade by locating and correcting the subtle grammar error in the poem. (We leave this as an exercise for the reader.) Those are the basic parts of speech, just as we learned them in school. How does HaleyAuthority use them to interpret sentences?

2.2.2 Grammatical roles in a sentence

The parts of speech play various *grammatical roles* when used in a sentence, just as actors may play various roles when they appear on stage. A typical sentence offers a predictable set of roles for the parts of speech to play. Most sentences have a verb, a subject and a direct object, and there may also be complementary words and phrases.

In HaleyAuthority, the *subject* of a sentence is usually a noun. The subject of the sentence is the person, place, thing, or idea that is doing or being something. The subject always comes before the verb.

The applicant's occupation is snake charmer.

It is tempting to think the sentence is talking about the applicant, but actually the subject of the sentence is "occupation."

A sentence always has a *verb*. In HaleyAuthority, this is often a form of the verb *to be*, such as *is, was, will be, or should be*.

The applicant *is* 40 years old.

Some sentences tell a story. The subject tells you who did it. The verb tells you what happened. The *direct object* tells you to whom or to what it happened. You usually find the direct object right after the verb.

¹ King, Graham, *The Sunday Times Good Grammar in One Hour*, Mandarin Paperbacks, London, 1993, p. 35.

Philip mailed *a letter*.

If a sentence has a direct object, then it might also have an *indirect object*. The indirect object tells us *to whom* or *for whom* the action was directed. An indirect object usually refers to a person; a preposition always introduces it.

Philip mailed a letter *to Margaret*.

Always? Well, no. People often reverse the position of the direct and indirect objects and drop the preposition.

Philip mailed *Margaret* a letter.

This sentence really means “Philip mailed *to Margaret* a letter,” but that sounds awkward. HaleyAuthority is aware of these variations in phrasing and adjusts for them automatically. Variations such as this are routine.

A *complement* is a prepositional phrase that follows a noun or a verb. It often resembles an indirect object, but it refers to an inanimate object rather than to a person or other living thing.

John raced Howard *in the pool*. (Follows the noun “Howard.”)

A person resides *in a country*. (Follows the verb “resides.”)

When you define a phrasing in HaleyAuthority, you will identify the subject, the verb, and the direct and indirect objects. Then you will attach any leftover words by identifying which parts of the phrasing they complement. You will specify which word the complementary phrase should follow in the phrasing.

Identifying the grammatical roles in a phrasing is very simple in most cases. The challenge comes when we start to educate HaleyAuthority about the *meaning* of the words.

2.3 Business concepts

To understand your business policies, HaleyAuthority needs to understand a few things about your business. Understanding English is not enough. HaleyAuthority needs an underlying model of your business objects, their attributes, and their relations to one another.

For instance, you might write a statement that begins:

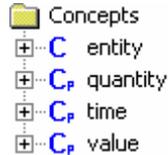
If an applicant’s age is more than 40 years...

To understand this phrase, HaleyAuthority needs to know that applicants exist, that applicants are people, that people have ages, and that age is a number of years.

HaleyAuthority did not know these things when you installed it. Adding this type of information is called *semantic modeling*. Semantic modeling lets you describe your business objects using HaleyAuthority's entities, values, and relations.

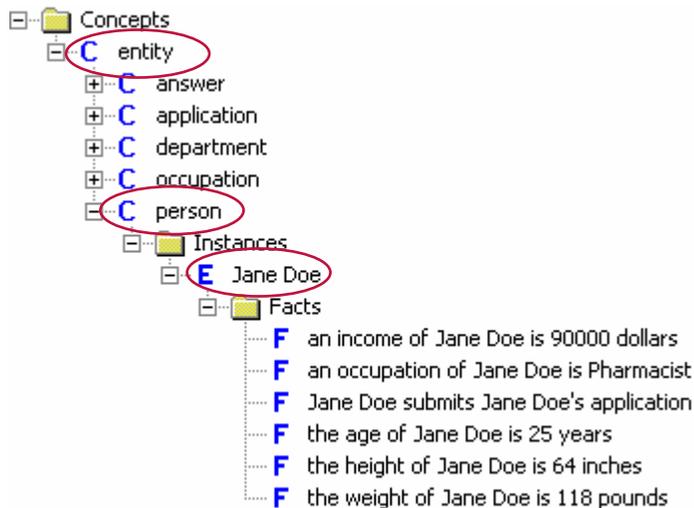
2.3.1 Entities

At the very top of the knowledge tree in HaleyAuthority's **Full View** tab you will see a node called **Concepts**:



The **entity** node contains complex objects (such as people), that have associated attributes (such as age). Your business object model will be built of these entities. The entity node is empty to begin with; you will populate it with your own entities.

For instance, we could say that a person is an entity and Jane Doe is an instance of a person. Defining these entities creates the following concept tree in HaleyAuthority:

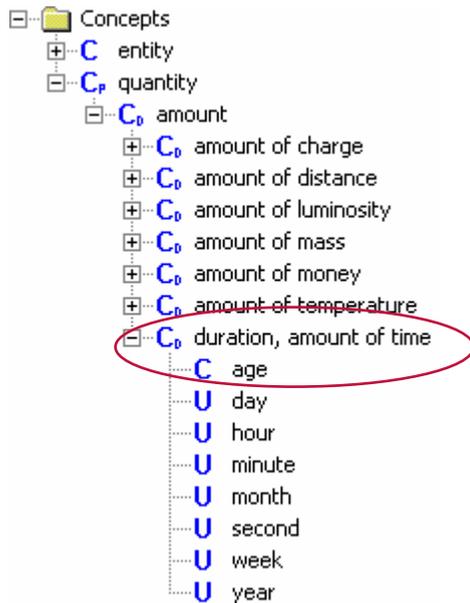


You can see that Jane Doe has several associated attributes, such as height, weight and age. These are *values* associated with the **person** entity or the **applicant** entity by relations.

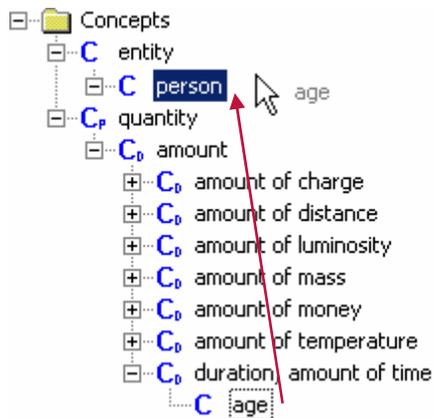
2.3.2 Values

HaleyAuthority has built-in knowledge of many types of values. It is important to explore the tree structures inside the **value** node to see what is available there. For instance, to

create a value for “age” we simply navigate to values expressed in “amount of time” and add a new kind, called “age.”



Then we use the mouse to drag “age” up the screen and drop it on “person”. Instantly, HaleyAuthority understands that “a person has an age,” and that we might be about to tell it something about “the age of a person.”



As you might have guessed, entities and values have names, and the names are *nouns*. Whenever you create a new entity or value, HaleyAuthority simultaneously creates the necessary noun and adds it to the **Dictionary** node of the knowledge tree.

2.3.3 Relations

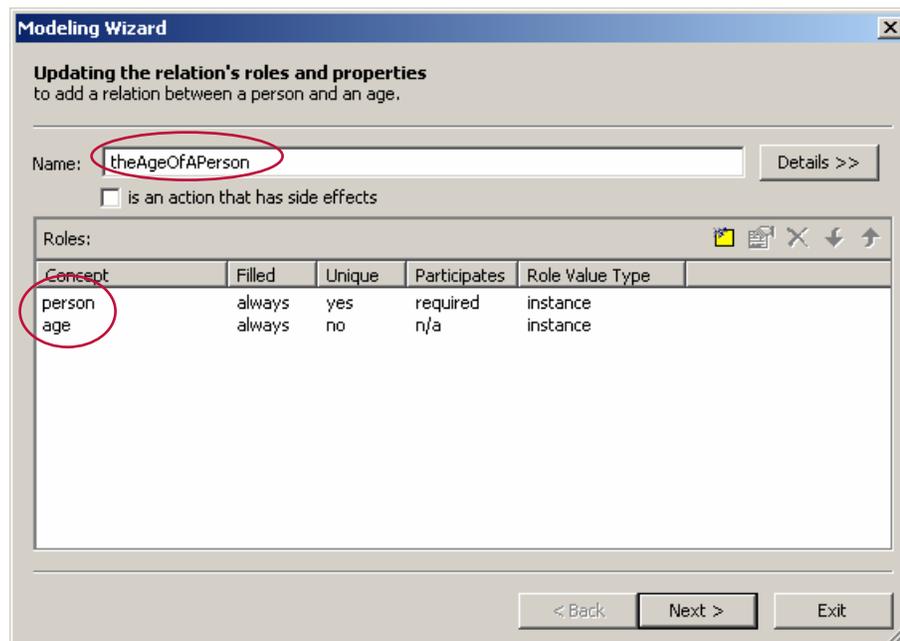
Relations connect values to entities, and connect entities to one another. The easy way to understand a relation is to peek under the hood for a moment and see how the Eclipse

rule engine represents a fact. We know that John Doe is 56 years old. The corresponding fact says:

```
(Person_age John_Doe 56 042)
```

In this simple statement, **Person_age** is the name of the *relation*. **John_Doe** is a symbol representing the *entity* John Doe. The following number, **56**, is the *value* of John's age in years. The second value, 042, indicates the unit of measurement, which is years. The **Person_age** relation, therefore, is a package that always contains the name of an entity (appearing in the semantic role of *applicant*) and a number (in the semantic role of *age*). There might be hundreds of such facts available in Eclipse.

Now we will see how this same relation looks in HaleyAuthority. This is the initial screen of the **Add a relation** modeling wizard. Notice that you get to name the relation yourself in the top field of the box (**theAgeOfAPerson**). In the **Roles** section of the screen is a list of concepts (entities and values) that participate in this relation. You can see that **theAgeOfAPerson** relates the **person** entity to the **age** value. To define a relation, you type in the new relation name and then select the entities and values from lists. It is very simple.

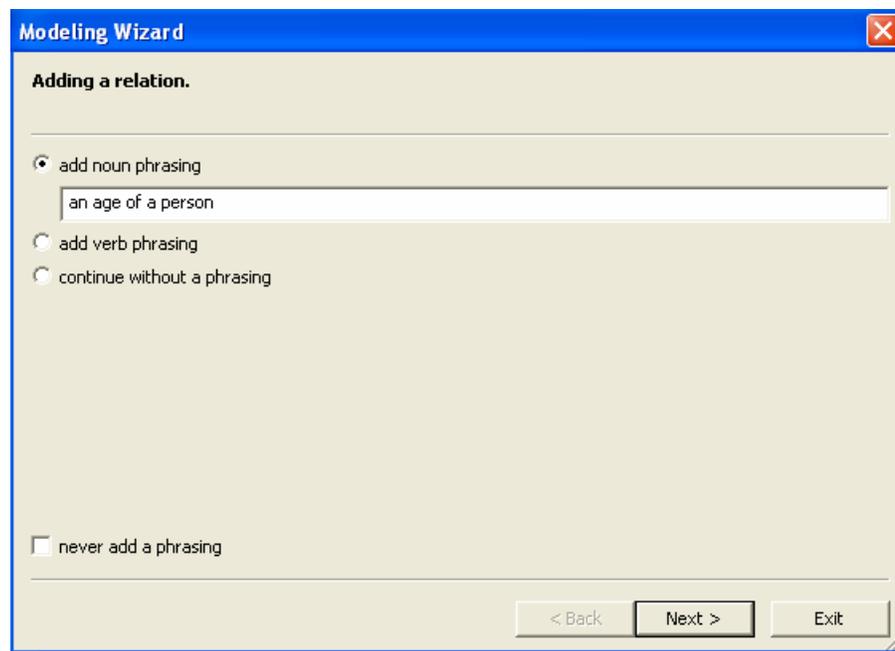


There are quite a few pre-defined relations in HaleyAuthority, and you will undoubtedly define many more of your own. Relations are perfectly simple except for one thing – HaleyAuthority needs to know how the nouns, verbs and other parts of speech in a policy statement will map into the semantic roles of the relation. To make this bridge each new relation requires one or more *phrasings*.

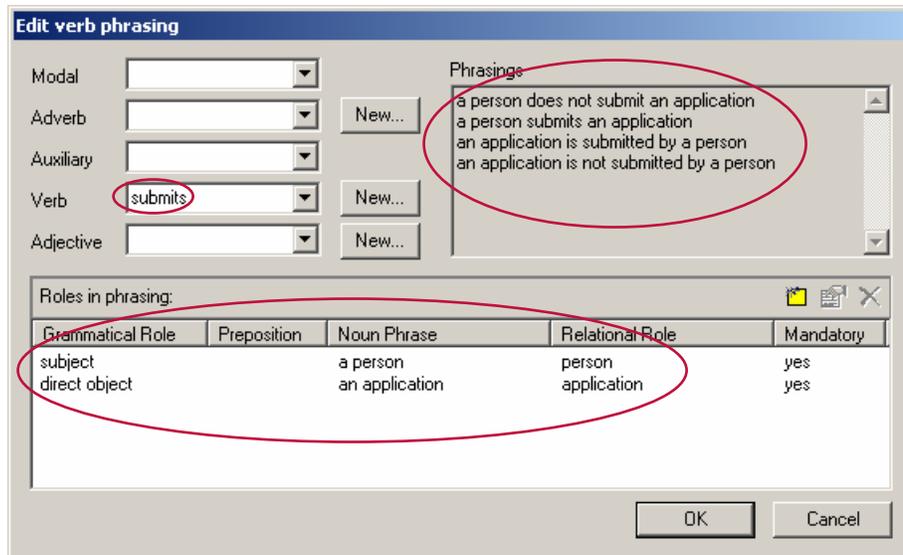
2.3.4 Phrasings

Before HaleyAuthority can use a relation in a meaningful way, it needs one or more examples showing how the relation is expressed in English sentences. These examples are called “phrasings.” A typical verb phrasing has a subject, a verb, and a direct object, as in “a person (subject) has (verb) an age (direct object).” The phrasing may also contain other significant parts of speech such as adverbs, auxiliary verbs, and noun and verb complements.

There are two types of phrasings in HaleyAuthority, noun phrasings and verb phrasings. Noun phrasings in a relation usually indicate possessive or ownership, such as “a person’s age”. Whereas verb phrasings always include a verb, “a person has an age”. The relation listed above, theAgeOfAPerson, can be created via either a noun phrasing or a verb phrasing. If created as a noun phrasing, you directly enter the phrasing for “an age of a person”. With this phrasing HaleyAuthority will automatically understand that “a person has an age”, “an age of a person”, and “a person’s age”.



For verb phrasings, we don’t just type in phrasings. Instead, we select the critical entities, the values, and the verb from lists, and then have HaleyAuthority *assemble* them into the desired phrase. HaleyAuthority performs a sanity-check by displaying the English phrases it thinks we want. We often have to adjust our input to get the phrasing just right. For the new user, there is a certain amount of trial-and-error to be expected.



This is a phrasing for the **anApplicationOfAPerson** relation. We see that the subject of the phrasing is “a person.” The direct object is “an application.” The selected verb is “submits.” HaleyAuthority takes these pieces and builds a suggested phrasings: “A person does not submit an application,” “a person submits an application,” “an application is submitted by a person,” and “an application is not submitted by a person.”

Once the phrasing is correct, HaleyAuthority will understand English references to the objects in this relation, and may generalize in surprising ways. HaleyAuthority automatically extends the language model in appropriate directions to create the greatest *unambiguous* flexibility of expression. From the above phrasing, HaleyAuthority now understands “a person submits an application,” and “an application is submitted by a person.”

2.4 Statements and rules

Using entities, values, relations and phrasings, HaleyAuthority can take your statements of company policy, written in English, and translate them into data-driven rules for the Eclipse rule engine to execute. Incoming business data can be processed according to company policy, and correct decisions can be made at high speed.

How does that work?

An Eclipse *rule* is a simple statement of conditions and consequences, similar to an if/then sentence in English.

If <these conditions are met> then <perform these consequences>.

A *statement* in HaleyAuthority is the same idea expressed in English. HaleyAuthority takes English statements and turns them into rules behind the scenes. The Eclipse rule

engine maintains a collection of “facts” about the current situation. We say that a rule “fires” when all of its conditions are met by corresponding facts. A rule does *nothing* until the right facts become available. Then it fires *once* for *each* set of matching facts.

When a rule fires, the rule engine executes the rule’s consequences. The consequences may add or remove facts, or they may have side effects outside of Eclipse. Changes to the collection of available facts may cause additional rules to fire, a process known as *chaining*.

Let’s examine a simple example of data-driven chaining to see how a rule-based system typically works.

S an applicant has a hazardous occupation if the applicant is a scuba diver

This rule, “an applicant has a hazardous occupation,” does nothing as long as there are no scuba-diving applicants, as defined by the condition. Then John Doe, a diver, applies for insurance. Suddenly the system contains new facts about John Doe.

John Doe’s occupation is “scuba diver.” The rule about scuba divers matches this new fact and fires. The result is the creation of a new fact; John Doe has a hazardous occupation. There is another rule and condition in our system that is sensitive to applicants who have hazardous occupations.

S an application should be referred if the occupation of the person who submits the application is hazardous

This rule notices that John Doe has a hazardous occupation. It fires and creates another new fact; John Doe’s application should be referred.

There could be many rules that recognize situations where an application ought to be referred. The rules react to the incoming facts about John Doe. Any rule that fires makes a change in the collection of facts, adding a recommendation about how to handle John Doe’s application. These rules develop our knowledge about John Doe, but have no side effects outside of Eclipse.

Once the status of the application has been determined, action rules come into play. They wait patiently and only fire after receiving the recommendations of the other rules. Here are two action rules:

S approve an application if the application should not be referred

S if an application should be referred then refer the application to the Underwriting Department

In our example, the second of these two rules would fire. Instead of creating a new fact, this rule executes an external procedure that refers John Doe’s application to the underwriting department.

Rule-based systems are very simple to create and maintain compared to the tangled procedural code one usually finds in e-commerce applications. There may be hundreds or thousands of rules in the system, but each one reacts independently to a specific situation.

Every part of the system is as simple as “if/then.”

2.5 Scenario

Mike Marks was still a little doubtful. He could see how English sentences might map into operations on his business objects, but it still seemed too good to be true. Every new project had a hidden difficulty, and it was his job to find it. It would be better to find it early rather than late.

What if the software really could understand policies written in plain English sentences? That would be a great step forward, but there had to be a catch. Mike frowned and reached for the company’s employee handbook. He turned to the new section on vacation policy and read a few paragraphs. He smiled grimly and dropped the book in the wastebasket. He had found the catch.

“Policy makers can’t write plain English,” he grumbled.

3 Writing policies clearly

HaleyAuthority can understand your policy instructions in plain English. The software is very intelligent about exploiting the English language to create rules for the Eclipse rule engine.

English, however, is often imprecise. Humans routinely apply common sense to ambiguous sentences to determine their meaning, which is a level of reasoning that computers have not yet achieved. Therefore, even though HaleyAuthority understands your English statements, the statements must be written as plainly and clearly as possible.

Most people are not accustomed to writing clear, unambiguous English. They rely instead on the reader's judgment. The knowledgebase administrator may need to do some interpreting and rewriting before typing the statements into HaleyAuthority.

3.1 Scenario

Having determined that a rule-based system was a good choice for the proposed Automated Application Approval system, Mike Marks went to visit the head of the underwriting department, John Stevens, for some sample approval policies. Stevens was able to quote a few guidelines from memory, but did not know the complete list. "I'll have one of my people send you the complete manual," Stevens promised.

Mike returned to his office with his notes. Stevens had dictated a short list of application-approval guidelines:

- 1. If the applicant's age is less than 18 or more than 40 then refer him to underwriting.*
- 2. Non-smokers are not usually referred to underwriting.*
- 3. We do not refer applicants who don't have hazardous occupations.*
- 4. If the applicant's coverage is less than 80% of his income then the policy might be eligible.*
- 5. If every health question on the application was answered "no" then the policy might get automatic approval.*

6. *If the build table height and weight are within normal limits then the applicant might be eligible.*

Mike looked up from his notes and frowned. "Might be eligible?" he said.

3.2 Clarifying the statements

A computer requires simple, unambiguous instructions. The six policies stated above are neither simple nor unambiguous. We need to analyze and simplify them before "teaching" HaleyAuthority how to approve insurance applications.

Clarifying raw policies is an adventure in disciplined thinking. Always keep in mind that the computer obeys your instructions literally. Therefore, you have to eliminate every source of ambiguity in your statements.

3.2.1 Find the unstated assumptions

Human communication is full of implicit assumptions. For instance, you certainly noticed some problems with the raw policies. Did you notice that there was no description of the *action* the system is expected to perform? This is the first thing we need to know to build a working system. What *decision* are we making here?

After you analyze the policies, it should be clear that the underwriters expect an application to be approved *unless* the application violates one of their referral policies. This is a critical rule that the policy makers left unstated.

A new application will be automatically approved, *unless* the application violates an approval policy.

What should happen if the application violates a company policy? That is a second hidden assumption, and completes the set of action rules for this simple system.

If an application violates a policy, then it will be referred to an underwriter.

These two statements are not in their final form yet, but we have them written down and included in the list. We know now that the analysis of each application will end in an *approval* or a *referral*.

We will set up the system so that these two action rules are prevented from firing until all other rules have made their recommendations.

3.2.2 Identify the concepts

Each time the system runs, it will approve or refer a... what? Looking at the policy memo, it seems that we might approve an "application," an "applicant," a "non-smoker," or an insurance "policy." The author of the six statements was not very consistent.

You can educate HaleyAuthority to understand that an *application*, an *applicant*, a *smoker*, and an insurance *policy* could all be names for the same approvable object, but that isn't the correct approach. It is better to give the approvable object a standard, intuitive name. This makes it much easier for HaleyAuthority and the policy makers to communicate clearly.

For instance, the AA Application system will approve or refer *applications*. It quickly becomes apparent, however, that the approval policies all talk about *applicants* instead. The policies refer to the applicant's age, the applicant's weight, the applicant's salary, and so forth. These are intuitive English phrases that appeal to the policy makers. Therefore, the AA Application system will need to include an **applicant** concept and an **application** concept, at a minimum.

All objects in the system need to have these intuitive labels. If the labels are awkward or arbitrary, like the variables used in Java code, the policy makers will not be able to interact with HaleyAuthority's statements.

3.2.3 Minimize negative logic

HaleyAuthority understands "not" in a variety of contexts, but the overuse of negative statements invites confusion between HaleyAuthority and the policy makers. Consider this approval policy:

We do not refer applicants who don't have hazardous occupations.

As it stands, the sentence implies that an applicant in a safe occupation will never be referred to underwriting, *no matter what else we might know about him*. That can't be right. What did the policy's author really intend? Let's remove the negatives and rewrite the sentence more clearly:

We refer applicants who have hazardous occupations.

This policy is much more clear, and doesn't conceal an error in negative language. As a rule of thumb, try to rewrite statements that have more than one "not" in them.

3.2.4 Investigate limits

Policy makers are often careless about expressing limits and ranges. For instance, let's look at this policy again. What does it really mean?

If the applicant's age is less than 18 or more than 40 then refer him to underwriting.

A human reader might apply common sense and conclude that 18-year-olds should *not* be referred, but 40-year-olds *should* be.

HaleyAuthority won't see it that way. When the policy maker expresses a limit, always go back and clarify the boundary conditions. In this case the policy maker intended to refer applicants who were *less than 18 years old*, and also to refer applicants who were *at least 40 years old*.

If an applicant's age is less than 18 years or at least 40 years then refer him to underwriting.

Make sure you know how the border cases should behave. HaleyAuthority will help by automatically recognizing the phrases "less than," "more than," "at least," "at most," and "equal to."

3.2.5 Split or into multiple statements

If the conditions of the statement contain "OR," you can reduce confusion by separating the statement into two or more simpler statements. For instance, what if you encounter a policy that looks like this one?

If an applicant has a hazardous occupation or the applicant's age is more than 40 then refer him to underwriting.

This statement contains two policies masquerading as one. HaleyAuthority will insist that you separate these independent "OR" conditions into two statements. Don't be surprised when you try to type "or" in a statement and HaleyAuthority won't cooperate. An elegant statement describes exactly *one* situation. We don't want to try to code complex conditional logic in English.

What about this statement then?

If an applicant's age is less than 18 years or at least 40 years then refer him to underwriting.

This isn't quite the same thing. Technically this statement has a *single condition* that contains multiple numeric tests on one parameter. We could split this statement into two statements but it isn't necessary. HaleyAuthority has no objection to "OR" in this context.

3.2.6 Minimize punctuation

HaleyAuthority isn't interested in most punctuation marks. It doesn't permit commas, periods (full stops), or question marks in statements. However, parentheses are permitted for use with mathematical operations.

HaleyAuthority does recognize the possessive apostrophe, however. Instead of saying,

If the age of an applicant is...

you can say,

If an applicant's age is...

This is a very convenient feature that provides a natural way to express conditions based on objects and their attributes.

3.2.7 Minimize plurals

Policy authors often use plural nouns unnecessarily. It is important to remember that Eclipse rules generally match only *one object at a time*.

Therefore, we can avoid a source of confusion by stating our policies in terms of single, discrete objects. For instance, there is no need to speak of multiple applicants and occupations in this policy:

Refer applicants who have hazardous occupations.

This is better:

Refer an applicant who has a hazardous occupation.

Rules automatically evaluate all available matching objects, so you don't have to insist that a policy applies to all applicants. When you refer to "an applicant," the rule will evaluate *every applicant* it can find, but it will evaluate *each one separately*.

As another rule of thumb, avoid plurals unless the plural form is exactly what you mean and the singular form just won't do. This situation arises when you want to count the total number of something. HaleyAuthority responds appropriately to a statement like this one:

If there are less than 10 passengers, then cancel the flight.

Counting "passengers" is a legitimate use of a plural noun in a statement.

3.2.8 Avoid mass nouns

HaleyAuthority wants to know *how many*, not *how much*. When you talk about items that can be counted, you are using *count nouns*.

HaleyAuthority works well with objects that can be counted: one applicant, two applicants; one policy, two policies.

A *mass noun* does not lend itself to counting. An example would be "coverage." We don't say "one coverage, two coverages." Instead we speak of "some" coverage. "Some" is too vague for HaleyAuthority to understand.

Fortunately, mass nouns can be converted into count nouns by rephrasing to include units, such as units of mass or volume. The mass noun "coverage" can be converted to a count noun by speaking of an "*amount* of coverage," (one amount, two amounts).

HaleyAuthority can work with that. Similarly, HaleyAuthority would prefer to deal with a *gallon* of gasoline, a *pound* of butter, and a *dollar* of revenue.

Think of this as an *ounce* of prevention.

3.2.9 Avoid pronouns

Pronouns are the unbound variables of the English language: *he, him, her, who, them, they, it*. When people read a sentence containing pronouns, they unconsciously make assumptions about the meaning of each pronoun. Some of this interpretation is based on common sense, which a computer does not share. For this reason, pronouns introduce ambiguity.

Consider this sentence: "Bill insulted Phil, who scolded him." The human reader uses common sense to determine who was scolded.

The computer doesn't have the background knowledge to decide who scolded whom.

For this reason, it is better to eliminate pronouns from your HaleyAuthority statements.

This one needs to be rewritten again:

Refer an applicant who has a hazardous occupation.

It is better to be explicit:

Refer an application if the occupation of the person who submits the application is hazardous.

The sentence is not quite as graceful as before, but its meaning is crystal clear.

3.2.10 Using *a, an, and the*

HaleyAuthority pays close attention to the *determiners, a, an, and the*. It makes a special distinction between *a/an* object and *the* object. This distinction leads to HaleyAuthority's most common beginner's error.

- When you write a condition about *a* or *an* object, such as *an application*, HaleyAuthority applies the statement to *every available object* of that type.
- If the condition mentions *the* object, such as *the application*, HaleyAuthority thinks you mean *one specific object*, and you expect HaleyAuthority to know which one you mean.

Generally speaking, it is good practice to phrase your HaleyAuthority statements like this:

If *an* applicant is a smoker then *the* applicant's application should be referred.

This statement begins with “an” applicant, meaning the rule will match each available applicant who is a smoker. Then it continues with “the” applicant, meaning that we are still talking about the *same* applicant.

The rule will fire once for each smoker, referring each smoker individually to underwriting. How is this version different?

If *the* applicant is a smoker then *an* applicant's application should be referred.

HaleyAuthority won't like this version of the statement for two reasons. First, the condition mentions one specific applicant (*the* applicant) but does not say which one. Second, the statement isn't specific about which of many possible applicants (*an* applicant) should be referred.

Under most circumstances, HaleyAuthority will refuse to accept this version of the statement.

As we will see later in this tutorial, there is an exception. It is possible to tell HaleyAuthority that the rules will never see more than one applicant and application at a time. In that case, all references to “the applicant” or “an applicant” would always refer to the *same object*.

This feature greatly simplifies the task of writing statements because it lets HaleyAuthority relax its strict ambiguity checking. We refer to these unique objects as “singleton” concepts.

What's wrong with this statement?

If *an* applicant is a smoker and *an* applicant's age is more than 40 years then *the* applicant's application should be referred.

This rule will fire if any applicant is a smoker and, at the same time, that same applicant or *any other applicant* is over 40 years of age.

Whose application should be referred? Do we know? HaleyAuthority doesn't.

The best policy is to begin the statement by talking about *a/an object*, and then continue with references to *the object* as long as you mean the same object as before.

If you need to write a statement about two different applicants, note that HaleyAuthority understands references to “the first applicant” and “the second applicant.” This versatility is built in. For instance, you could write a statement like this one:

if an applicant is a smoker and a *second* applicant is not a smoker then the *first* applicant should go outside

3.2.11 Use present tense

To minimize confusion, make a habit of writing your HaleyAuthority statements in the present tense. HaleyAuthority is aware of the differences among verb tenses, and casual use of multiple tenses will force you to define multiple phrasings that do not add any value to your system. You will do extra work for the same result. You are welcome to introduce verb tenses if they are pertinent to the statements you need to define. The present tense is simply the easiest to implement and causes fewer misunderstandings.

3.2.12 Minimize modal verbs

There is a world of difference between saying “refer the application” and saying “the application *should be* referred.” One statement is an action. The other is just a recommendation.

The modal verbs are the ones that say *can, could, may, might, must, shall, should, will* or *would*. They introduce an intermediate level of truth into the system. This requires additional rules to perform actions based on the accumulated modal information.

We will use a level of modal reasoning as we implement the AA Application system because it is appropriate to the semantics of the application.

The rules in this example will recommend that an application “should” be referred to an underwriter, but an action rule will perform the actual referral.

In general, however, casual use of modal verbs complicates your system without adding any value. Keep them to a minimum.

3.2.13 Minimize *then*

We normally think of rules as if/then statements.

“*If* <these conditions are met>, *then* <perform some action>.”

HaleyAuthority understands your statements in the opposite order, too.

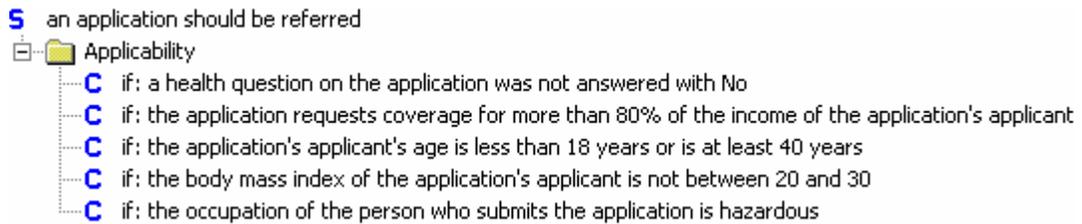
“<Perform some action> *if* <these conditions are met>.”

This format puts the “if” in the center of the statement and eliminates the “then.”

The difference between these two formats is largely a matter of style, because both statements compile into the same rule in Eclipse. The second format, with “if” in the center of the statement, has the virtue of alphabetizing similar consequences together in a list of statements.

If you have many statements in a module, this can be a convenient convention. We will take this approach in the following examples.

A more advanced approach is to use HaleyAuthority’s “applicability conditions” to provide a more formal structure to your collection of statements. For instance, we could organize multiple referral conditions into a single statement, like this:



This approach organizes and structures the knowledge formally, making it easier to locate an existing statement or to determine that it does not exist in the system.

3.3 Scenario

Mike Marks looked at the revised approval policies on the monitor. It had taken him an hour to simplify the underwriting guidelines to eliminate all ambiguity. He reviewed them one at a time.

Original: if the applicant’s age is less than 18 or more than 40 then refer him to underwriting

Revised: refer an application to underwriting if the applicant's age is less than 18 years or is at least 40 years

The revised version is a little more precise.

Original: non-smokers should not to be referred to underwriting

Revised: refer an application if a smoker submits the application

The revised version eliminated the double negation:

Original: we do not refer applicants who don’t have hazardous occupations

Revised: refer an application if the occupation of the person who submits the application is hazardous

The problem is, how do we know if the applicant has a hazardous occupation? It turned out that underwriting had a list of hazardous occupations. Mike set it aside for later attention.

The next policy had to do with the applicant’s requested coverage and income.

Original: if the applicant’s coverage is less than 80% of his income then the policy might be eligible

Revised: refer an application if the applicant requests coverage for more than 80% of the applicant's income

The next policy turned out to be simple.

Original: if every health question on the application was answered 'no' then the policy might be eligible for automatic approval

Revised: refer an application to underwriting if a health question on the application was not answered with no

Mike had to spend some time on the final policy.

Original: if the build table height and weight are within normal limits then the applicant might be eligible for automatic approval

He went back to the supervisor of the underwriting department to obtain the "build" table. It was dated and contained many gaps. Rather than input the table into HaleyAuthority, Mike dug a little deeper to find the formula the table was based on. The formula used the applicant's height and weight to calculate a score called the "body mass index." A body mass index score between 20 and 30 was acceptable.

Revised: refer an application if the applicant's body mass index is not between 20 and 30

After some experimentation, Mike determined that HaleyAuthority could calculate the body mass index using these two additional, simple rules:

a person's body mass index is the person's weight times 703 divided by the square of the person's height

the square of the person's height is the height times the height

The original six policies, when simplified and clarified, yielded eight statements. These statements detect all situations that "should" result in referring the applicant to underwriting.

Mike noted that six of the statements resulted in one action: refer the application.

Therefore, he regrouped the statements under the action to simplify the list:

An application should be referred if:

the applicant's age is less than 18 years or is at least 40 years

the applicant is a smoker

the occupation of the person who submits the application is hazardous

the applicant requests coverage for more than 80% of the applicant's income

a health question on the application was not answered with no

the applicant's body mass index is not between 20 and 30

Once these rules were done, however, Mike needed two action rules to finish up. These rules would run at a lower priority than the others, to be sure they didn't fire prematurely.

approve an application if the application should not be referred

if an application should be referred then refer the application to the Underwriting
Department

"Now," Mike thought, "all I have to do is type these statements into HaleyAuthority and I'm done." He reached for the mouse and clicked the HaleyAuthority desktop icon...

4 Educating HaleyAuthority

This chapter presents a step-by-step illustration of a knowledgebase administrator entering business policies into HaleyAuthority. It guides you through the details of navigating HaleyAuthority, while demonstrating how to think about the tasks that present themselves.

HaleyAuthority offers many powerful features and options, but you do not need to learn them all to get started. The essential procedures are few, and you can master the controls easily. The example really works; you can follow along if you wish.

When we finish this chapter, you will know how create statements with HaleyAuthority.

If you get lost at any point or need a reference while creating this example, a completed version of the tutorial knowledgebase ships with your HaleyAuthority installation. This example knowledgebase can be found under the *Examples/Tutorial* folder.

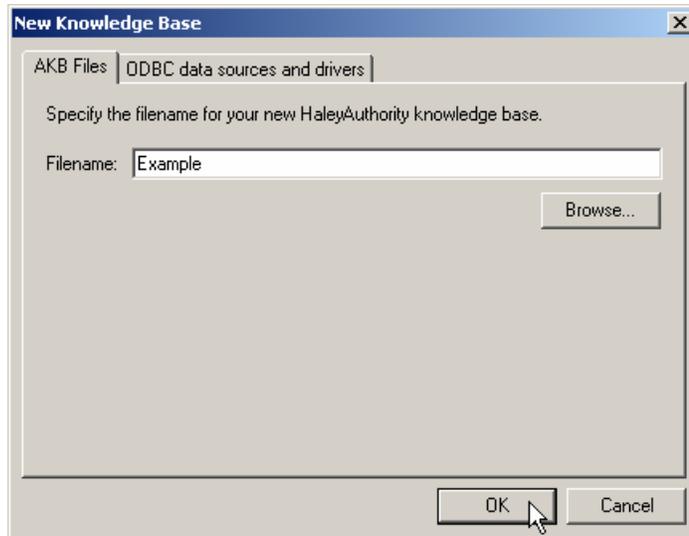
4.1 Scenario

Mike opened a knowledgebase in HaleyAuthority and picked up his list of proposed statements. He looked from the list to the screen. How do you start? He hadn't defined any entities yet, so would HaleyAuthority even let him type in a statement? He started expanding the various nodes of the tree to try to find out...

4.2 Creating a new knowledgebase

When you start experimenting with HaleyAuthority, it is a good idea to create a new, empty knowledgebase. This eliminates the distractions and interactions of pre-existing data, and avoids the problem of damaging someone else's work. Run HaleyAuthority. If it opens a knowledgebase automatically, use the **File** menu to **Close** that knowledgebase. You want the empty HaleyAuthority window with no knowledgebase open.

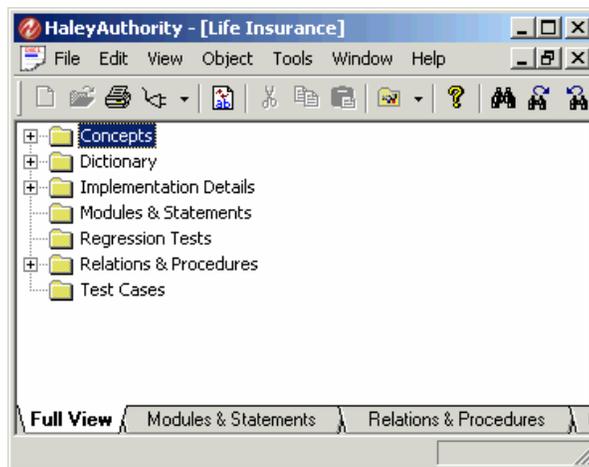
Click **File** and **New...** to create a new knowledgebase. You will see this dialog:



The **AKB Files** tab refers to "HaleyAuthority Knowledge Base Files." Type in the name of the new file and click **OK**.

HaleyAuthority will use your Microsoft Windows user name as the default user name.

It takes a few seconds to initialize a new knowledgebase. When the data structures are in place, HaleyAuthority looks like this:



You can see the main nodes of the knowledge tree in the upper left, and several tabs along the bottom of the screen. The tabs provide an alternate means of navigating HaleyAuthority, similar to opening nodes of the tree. For the purposes of this chapter, we will stay in the **Full View** tab.

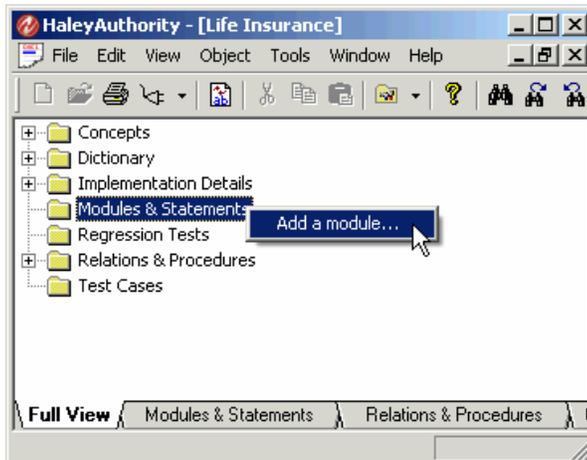
Now, where do we begin?

4.3 Modules, statements, and applicability conditions

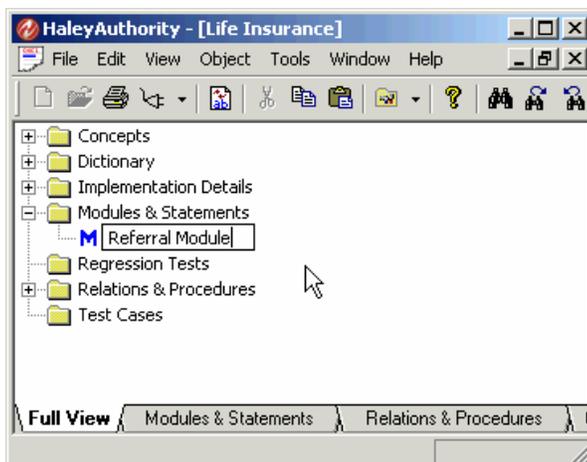
Rules are entered as English “statements” in HaleyAuthority. Similar or related statements are grouped into “modules.” You should think of modules as file folders, used for organizing your collection of statements. Modules of statements can have higher or lower priorities, can be turned on and off programmatically, and have other special features that let you organize hierarchies of statements. For instance, modules can have owners who control access to the module’s statements. You can move statements from one module to another if you need to, just like moving files between folders.

4.3.1 Adding a module

In the present context, we need a module for the statements that will refer an applicant to the underwriting department. Right-click the **Modules & Statements** node and select **Add a module...**



Give the module an appropriate name, such as “Referral Module” and click **OK**. You can change the name of the module later if you desire.

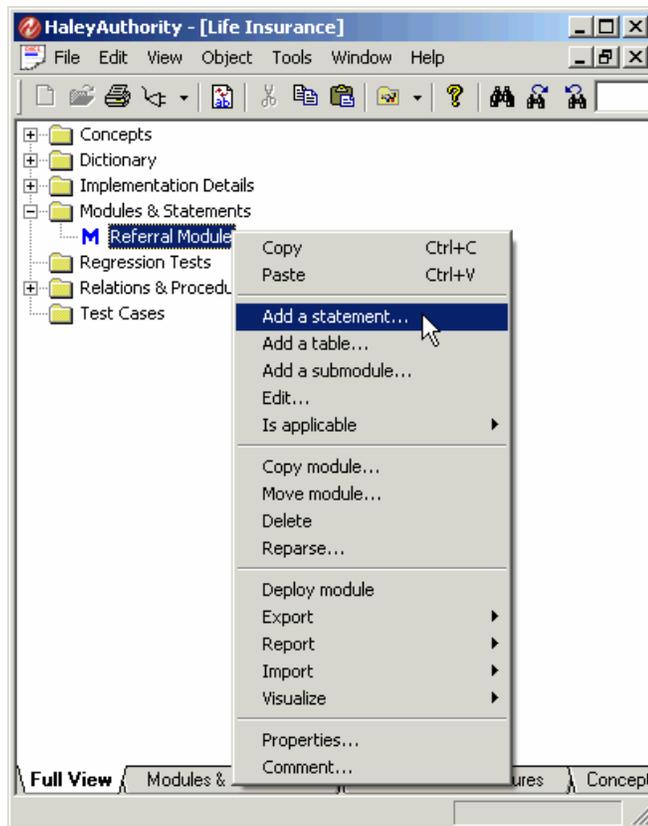


4.3.2 Adding a statement

The easiest place to start would be to tackle the statement and conditions that would trigger the referral of an application.

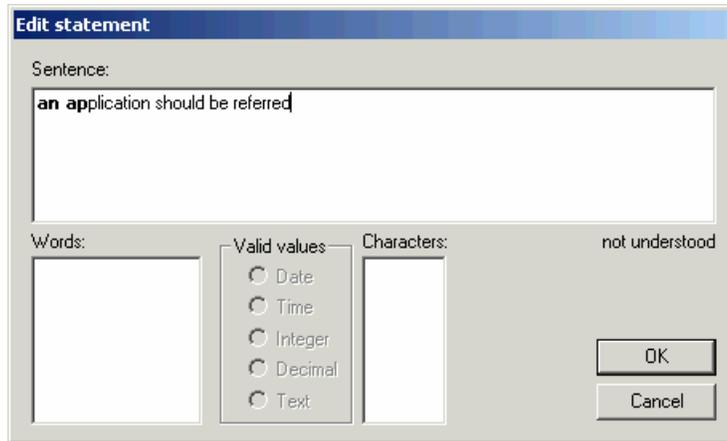
We begin by just typing it in, but wait... *Where* do we type it in?

Right-click the new module node and select **Add a statement...**



The **Edit statement** dialog opens.

To enter the first statement, simply type the statement directly into the top field of the **Statement Dialog**:



HaleyAuthority lets you type in the statement whether it understands you or not. This means that you can type in your statements just as easily as writing email. Then the fun begins. As the knowledgebase administrator, you have to be sure that HaleyAuthority *understands* the new policy.

You will notice that the first word of the statement, “**an**,” appears in bold face. The next word, “application,” appears in normal face.

Working from left to right, HaleyAuthority bolds as much of the sentence as it understands, and then reverts to normal face when it encounters a difficulty. The first difficulty is that HaleyAuthority doesn’t know what an “application” is. We’ll have to tell it.

Click **OK** to save the statement. We’ll come back to it in a minute.



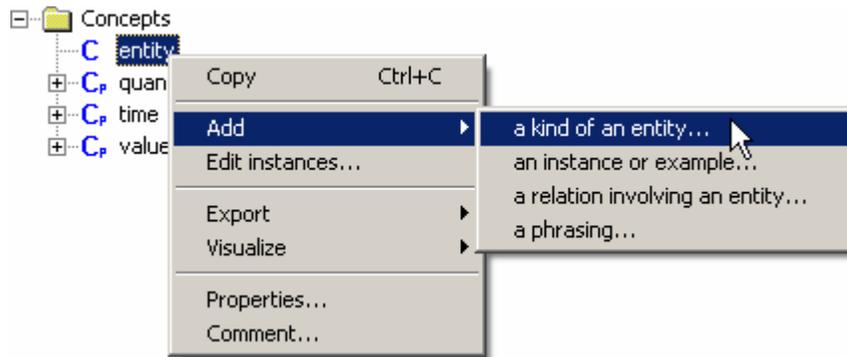
By the way, in the list of statements, the icon **S?** means that this statement is a draft and is not currently understood by HaleyAuthority. Statements understood by HaleyAuthority are preceded with **S**, without the question mark.

4.3.3 Adding an entity and a noun phrase

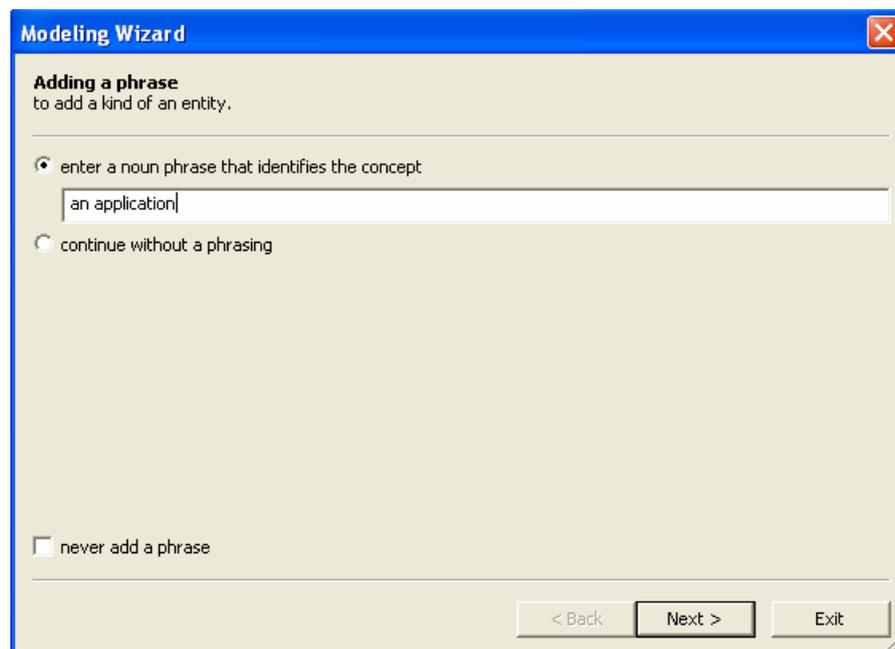
We need to do a little “semantic modeling” to tell HaleyAuthority what an “application” is. We don’t need to explain very much at this time, only that there is such a thing as an application, and that we expect to see only one of them at a time.

What kind of object would an “application” be? The choices are “entity” or “value.” An application is something important. It is likely to have characteristics. We intend to make decisions about it. Clearly this is not a value, which is a single, labeled attribute like “temperature.” Complex objects with multiple attributes are generally modeled as entities.

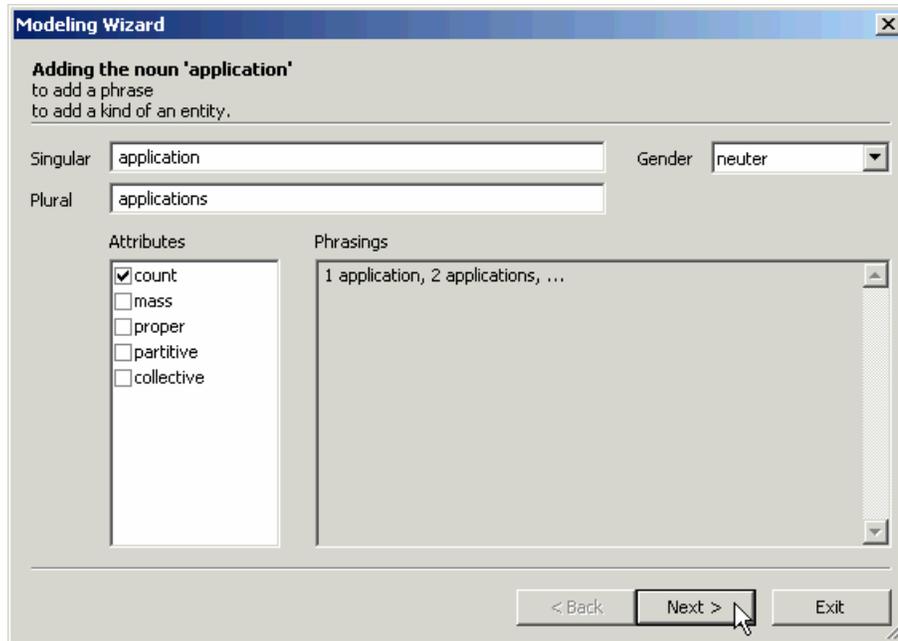
Open the **Concepts** node and right-click the **entity** node. Select **Add > a kind of an entity**.



The **Adding a phrase** modeling wizard opens. You will use this modeling wizard to add the noun and noun phrasing for “application.” Enter “the application” in the **enter a noun phrase that identifies the concept** field and then click **Next >**.

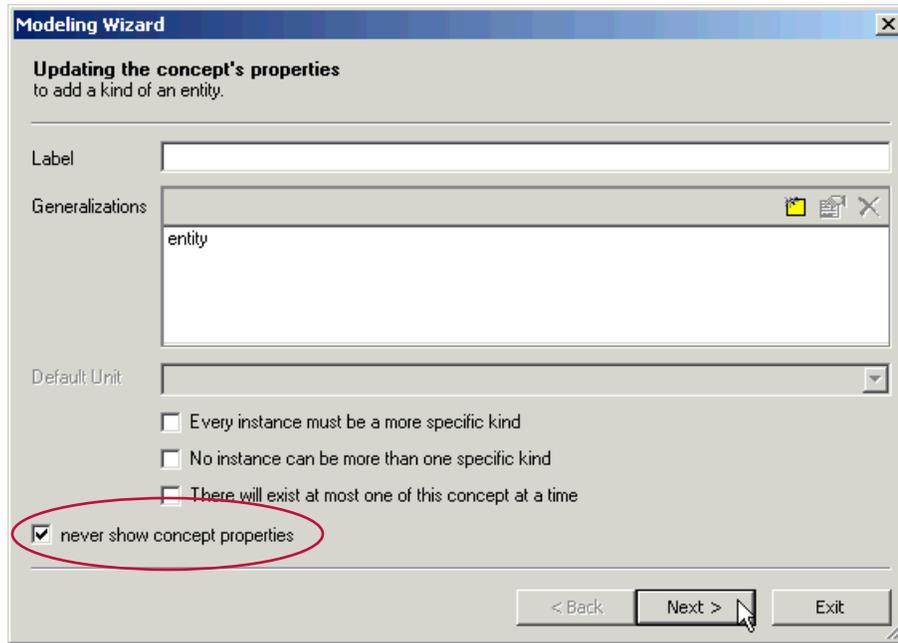


The modeling wizard displays the **Adding the noun 'application'** screen.



While in this screen, you should experiment briefly with the **Gender** field. The gender of a noun tells HaleyAuthority which pronouns might apply to it in a statement. Masculine nouns are often associated with masculine pronouns such as “he,” “his,” and “him.” Feminine nouns may be associated with “she” and “her.” Neuter nouns may be referenced by impersonal pronouns such as “it,” “its,” “that” and “which.” The “personal” gender is a special class where the noun refers to some living being of unknown sex. HaleyAuthority associates this noun with pronouns such as “who,” “whom,” and “whose.” If we used a pronoun to refer to an application, the pronoun would be “it” rather than “he,” “she,” or “who.” Therefore, the gender of this noun is neuter. It is a count noun, and HaleyAuthority has guessed the plural form correctly. Click **Next >**.

The modeling wizard displays the **Updating the concept's properties** screen. You do not need to make any edits to this screen of the modeling wizard. In fact, you may choose to have HaleyAuthority skip this screen for the duration of this HaleyAuthority session by checking the **never show concept properties**. Click on the checkbox to place a checkmark in the box and then click **Next >**.



The modeling wizard displays the **Updating the concept's implementation** screen. HalleyAuthority allows a concept to be implemented by a template that links to an external object, such as a Java class. You use this screen to specify the template. You do not need to make any edits to this screen of the modeling wizard; “application” will not be implemented by a template. In fact, you may choose to have HalleyAuthority skip this screen for the duration of this HalleyAuthority session by checking the **never show concept implementation**. Click on the checkbox to place a checkmark in the box and then click **Next >**.

The modeling wizard closes and HalleyAuthority adds **application** to the **entity** node.



If you reopen our statement, you’ll see that we have made some progress.

an application should be referred

HalleyAuthority now knows what an **application** is. It seems puzzled about what “should be referred” might mean.

4.3.4 Adding a verb phrase

How can we tell HalleyAuthority what “should be referred” means? Let’s look at the statement again.

HaleyAuthority doesn't understand that an "application" "should be referred." This pattern looks familiar. "Application" is the subject.

"Should be referred" is a verb (refer) using a modal (should) and an auxiliary (be). What are we looking at here?

When there is a *verb* involved, you need to create a new *relation*. Right-click the **Relations** node and **Add a relation...**

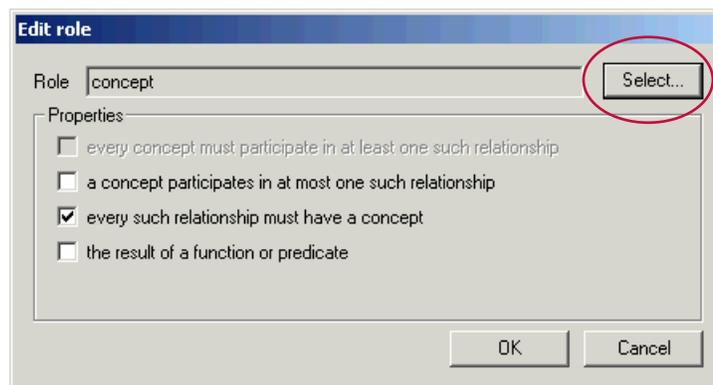


This opens the **Updating the relation's roles and properties** modeling wizard.

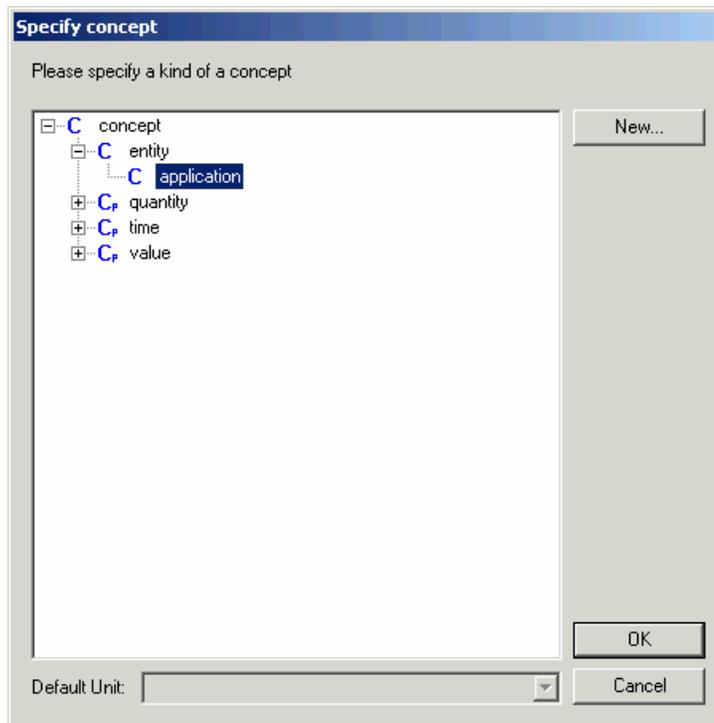
Start by typing in the name of the new relation. For this example, we can call it "anApplicationShouldBeReferred".

What entities are involved in this relationship? Actually there is only one entity, the **application**. We need to add it to the empty list of concepts in the **Roles** section of the screen. (This is actually a list of *entities* and *values*, which are both types of concepts.)

Click  in the **Roles** section of the modeling wizard. The **Edit role** dialog opens. There is nothing to do here yet, because first we have to **Select...** an existing concept.



Click **Select...**; the **Specify concept** dialog opens. Click [+]
to expand the tree control so that **application** is visible.



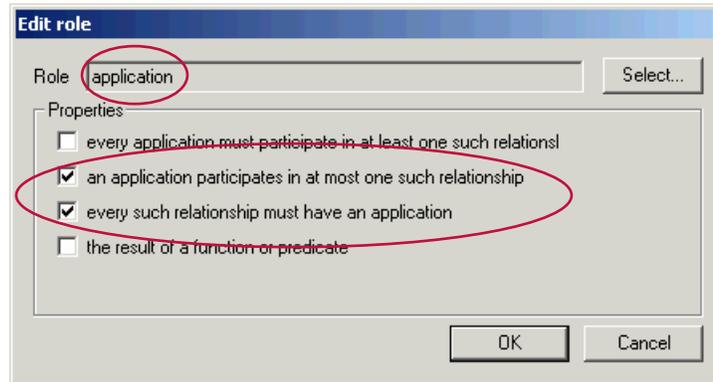
Select **application** and click **OK**.

This puts us back in the **Edit role** dialog, with **application** selected as the entity that fulfills the new role of the relation. At this point we have to stop and answer four questions:

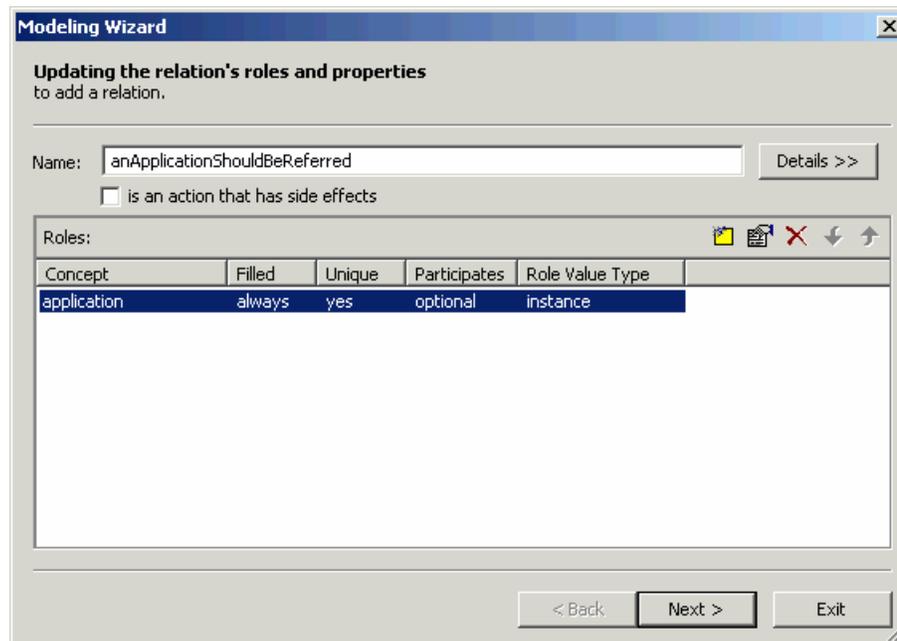
- The relation is “application should be referred.” Must every application participate in this relation? Clearly the answer is no. Some will not be referred.
- Does an application participate in *at most one* such relationship? It is a pretty simple relationship, which makes a recommendation about only one application. The answer is yes.
- Must every such relationship include an application? Yes. The relation would be meaningless without an application.
- Is the result of the relation a function or predicate? No. A function returns a value, such as the function we will define later to calculate the square of a number, as does a predicate, which returns a value of *true* or *false*. *anApplicationShouldBeReferred* will not return any value.

These questions detect differences in how the relation might be used. For instance, every son has a father. Not every father has a son. A father may have many sons. A son has exactly one father. HaleyAuthority needs this information in order to create Eclipse rule patterns that will match a single fact only, or match one of many related facts, or that can

match facts containing varying numbers of optional entities and values. Answer the questions one at a time and then click **OK**.



This takes us back to the **modeling wizard**. The **Roles** field now has an entry, **application**. The other items on the line simply summarize the selections you made in the previous dialog.



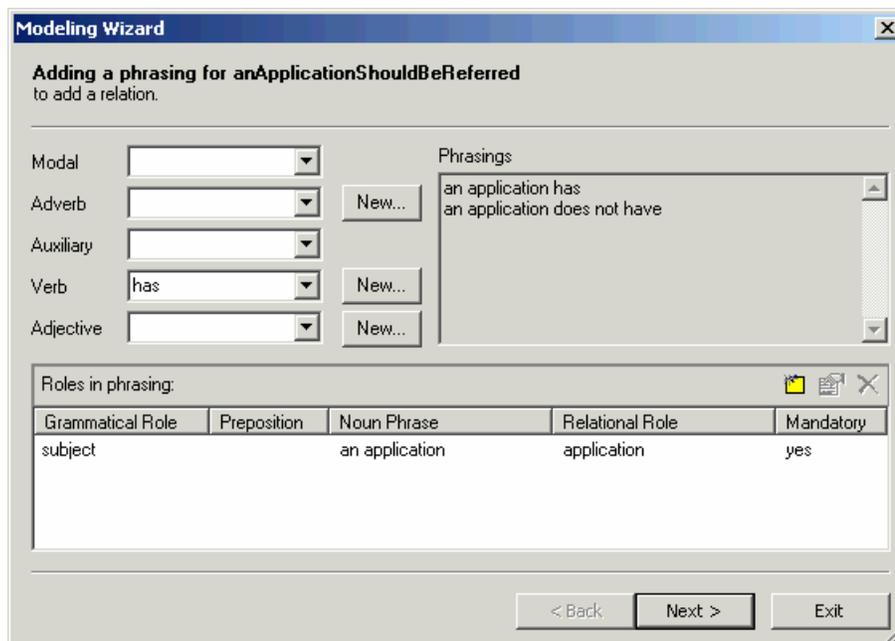
In this particular relation, **applicant** is the only entity or value involved. We don't have to add another role to this relation, so we can move on to creating a *phrasing*. In the future, however, you can use the following buttons when working with the concepts list (and similar lists in HaleyAuthority):

Button	Action
	Click to add a new concept
	Click to edit the selected concept
	Click to delete the selected concept
	Click to move the selected concept up the list
	Click to move the selected concept down the list

Click **Next >**. The **Add relation** screen of the modeling wizard opens. In this screen we will select **add verb phrasing**, because we still need to add the verb phrasing for the relation.

Click **Next >**. The **Adding a phrasing for [relation_name]** screen opens.

HaleyAuthority has correctly guessed that the application is the subject of the phrasing, but it isn't sure what comes next. What are we trying to say? "An application should be referred."

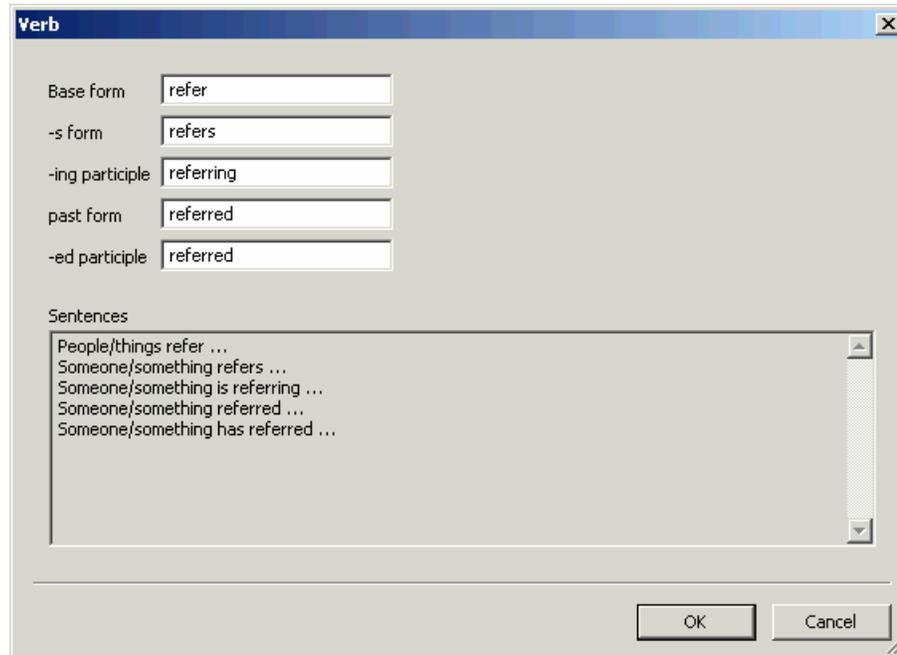


At this point, you should explore the lists in the upper left corner of the dialog. You'll find modal verbs, adverbs, auxiliary verbs, verbs, and adjectives. There are **New...** buttons next to some of the lists to let you create a new verb or modifier if you need one.

We can get most of the way just by shopping through the lists. Select "should" from the list of modals, and "be" from the list of auxiliaries. The missing word, "referred," is a verb.

We don't find *referred* in the list of verbs, so we'll have to create it. Click the **New...** button for verbs. This brings up the **Verb** dialog, which is pretty simple.

Type *refer* into the **Singular** field, and then check the other fields to see if HaleyAuthority has filled them in correctly. If it has, click **OK**.



The screenshot shows a dialog box titled "Verb" with a close button (X) in the top right corner. The dialog contains the following fields and text:

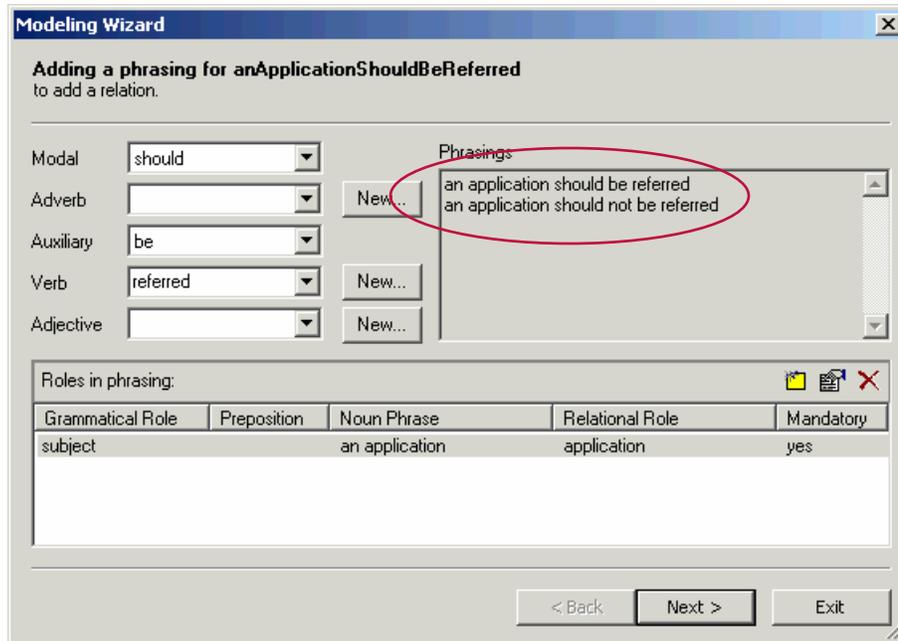
- Base form: refer
- s form: refers
- ing participle: referring
- past form: referred
- ed participle: referred

Below these fields is a list box labeled "Sentences" containing the following text:

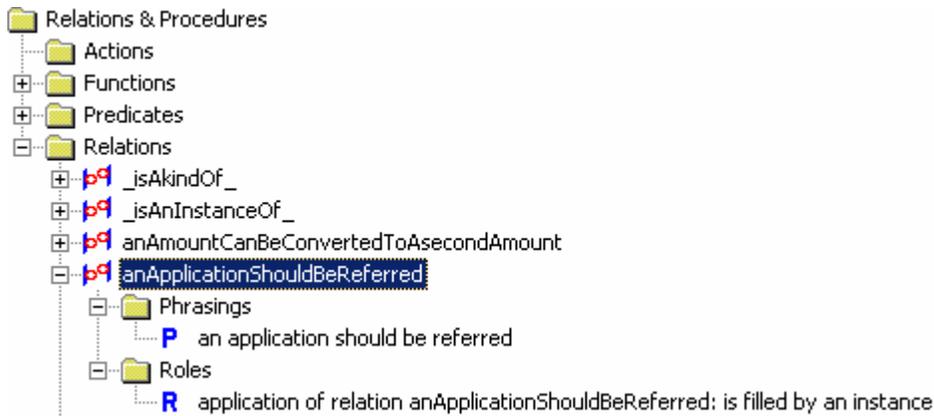
- People/things refer ...
- Someone/something refers ...
- Someone/something is referring ...
- Someone/something referred ...
- Someone/something has referred ...

At the bottom right of the dialog are two buttons: "OK" and "Cancel".

We are back in the modeling wizard. Look at the **Phrasings** section. Are the phrasings correct now? *An applicant should be **referring***? Pull down the list of verbs and find *referred* and select it. Now the phrasing reads, *An applicant should be referred*. Now the phrasing is correct. Click **Next >**.



This closes the modeling wizard (it closes now if you selected the options to not display concept properties and concept implementation during this HaleyAuthority session; otherwise, click **Next >** twice to close the modeling wizard). The new relation is added to the **Relations & Procedures** node.



Note that HaleyAuthority lists the phrasings and roles in the relation beneath the relation in the tree.

When we check our draft statement, we see that HaleyAuthority has made tremendous progress in understanding our intent:

an application should be referred

Click **OK** to save the statement as understood and close the **Edit sentence** dialog.

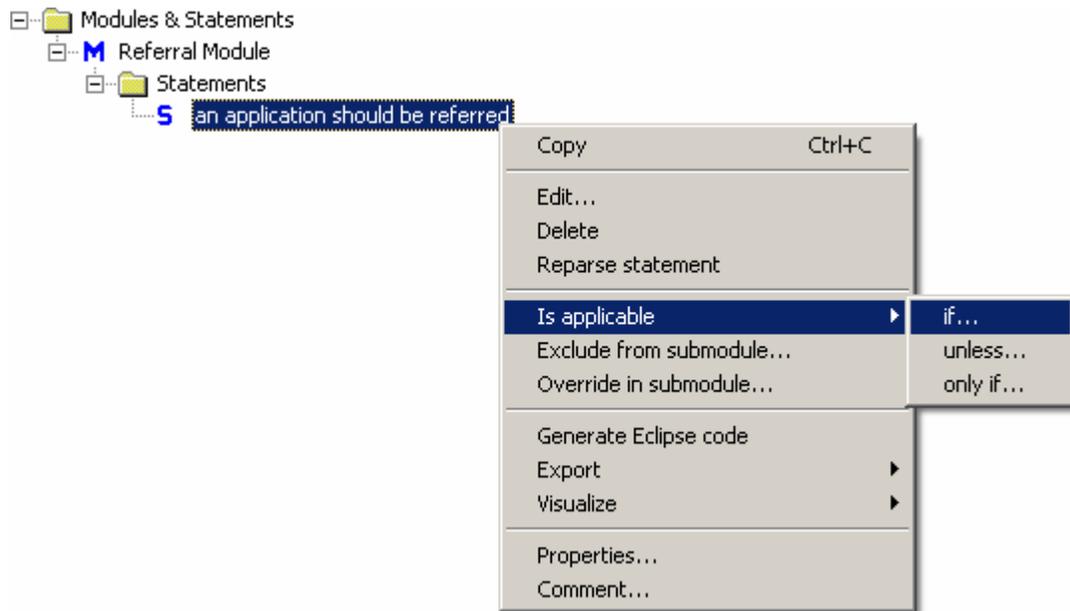
HaleyAuthority understands about an “application” and about “referring an application”.
Now we are ready to add the conditions to the statement.

4.3.5 Adding an applicability condition

The first applicability condition we will add is:

if a smoker submits the application

Right-click on the statement to which we are going to add a condition, **refer an application**, and select **Is applicable > if...** from the menu.



The **Edit statement** dialog opens. Enter the condition:

a smoker submits the application

HaleyAuthority understands only “a”, we will have to define the entities “smoker” and the verb phrase “a smoker submits the application”. (Do not enter the words **if**, **only if**, or **unless** in conditions; HaleyAuthority will automatically add the conditions.)

4.3.6 Adding the *person* and *smoker* entities

We need to tell HaleyAuthority what a “smoker” is. Be sure to think in terms of single objects. Define “a smoker”, not “smokers”. In the case of our insurance approval application, a smoker is a person who smokes and a person who submits an application. Smoker will be an entity, listed under the **entity** node of the knowledge tree. That much is clear.

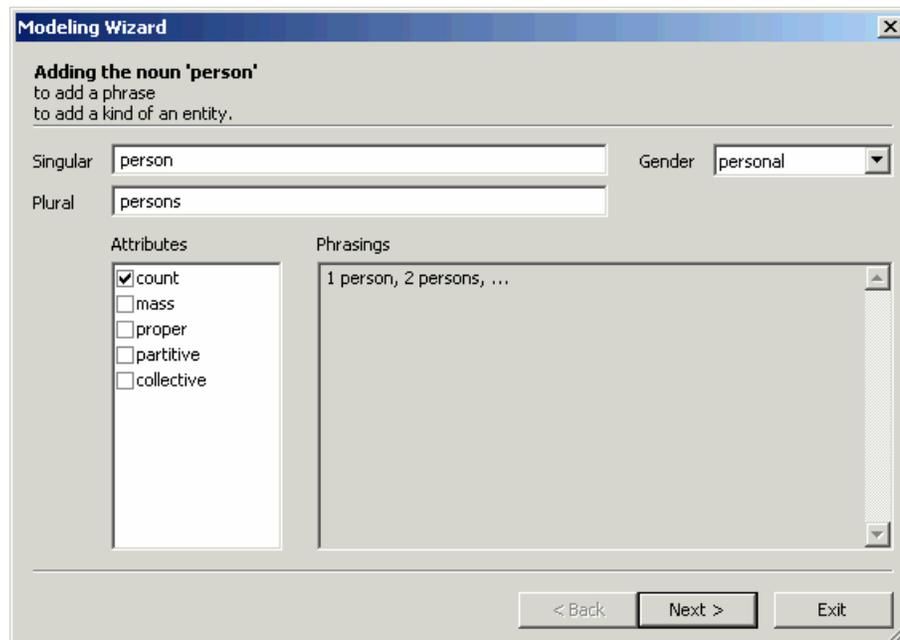
But a smoker is a person that submits an application. Because an applicant does not have any characteristics of its own, we would define the entity "person" and define the relation that "a person submits an application". By defining the noun phrase "the applicant of an application" for the "person" role of the relation, Authority understands that an applicant is a person who submits an application.

The first step is to tell HaleyAuthority that a "person" is a type of entity.

Right-click the **entity** node and follow the cascading menus to **Add > a kind of an entity...** The **Adding a phrase** modeling wizard opens. Type the noun phrase for the new entity, "the person", in the **enter a noun phrase that identifies the concept** field and then click **Next >**. The **Adding the noun [noun_name]** screen opens.

Do a quick sanity-check on the plural and phrasings fields. The proper plural of "person" is "persons," which is a *count noun*. You can count persons, as you see in the **Phrasings** field.²

For the noun "person," the *personal* gender seems appropriate. If we were to use a pronoun to refer to a person, we would say "a person *who...*"



The screenshot shows a dialog box titled "Modeling Wizard" with a close button (X) in the top right corner. The main title is "Adding the noun 'person'", with a subtitle "to add a phrase to add a kind of an entity." Below this, there are input fields for "Singular" (containing "person") and "Plural" (containing "persons"). To the right of the "Singular" field is a "Gender" dropdown menu set to "personal". Below the "Singular" and "Plural" fields are two sections: "Attributes" and "Phrasings". The "Attributes" section has a list of checkboxes: "count" (checked), "mass", "proper", "partitive", and "collective". The "Phrasings" section has a text area containing "1 person, 2 persons, ...". At the bottom of the dialog are three buttons: "< Back", "Next >", and "Exit".

When everything looks correct, click **Next >**. The modeling wizard closes and HaleyAuthority adds **person** to the **Concepts** node.

² Used as a plural, "people" is a mass noun (some people) that has no exactly corresponding singular form. You may use it as the plural of "person" if you wish.



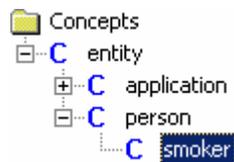
Now HaleyAuthority understands that a **person** is a concept. The **C** icon indicates a concept, by the way. That wasn't so hard. Now let's create a "smoker", who is a kind of a person.

Right-click **person** and **Add > a kind of a person...** The **Adding a phrase** modeling wizard opens. Type the noun phrase for the new entity, "the smoker", in the **enter a noun phrase that identifies the concept** field and then click **Next >**. The **Adding the noun [noun_name]** screen opens.

Do a quick sanity-check on the plural and phrasings fields. The proper plural of "smoker" is "smokers," which is a *count noun*. You can count "smokers", as you see in the **Phrasings** field.

For the noun "smoker", the *personal* gender seems appropriate. If we were to use a pronoun to refer to a smoker, we would say "a smoker *who...*"

When everything looks correct, click **Next >**. The modeling wizard closes and HaleyAuthority adds **smoker** to the **Concepts** node.

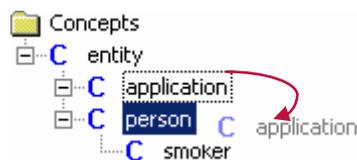


Let's check our condition. HaleyAuthority now understands smoker.

a smoker submits the application

We have already defined the entity application; our next step is to create a relation, *a smoker submits the application*. Because a smoker is a kind of **person**, we will use the generalization of **smoker**, which is **person**, to create the phrasing *a person submits the application*.

Click on **application** to select it and then drag it and drop it on **person**.



The **Updating the relation's roles and properties** modeling wizard opens.

HaleyAuthority has already listed the two concepts in the relation, **application** and **person**, in the **Roles** section of the wizard.

Type the name of the relation, *anApplicationOfAPerson*, in the **Name** field. Before we move on to define the phrasing, we must first check the properties of the concepts. Double-click on **person** in the **Roles** section. The **Edit role** dialog opens. Review the four properties questions:

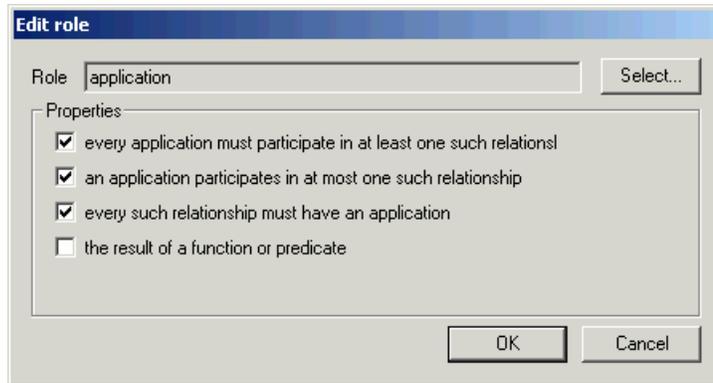
- The relation is “a person submits an application.” Must *every* person participate in this relation? Clearly the answer is no. Not every person submits an application.
- Does a person participate in *at most one* such relationship? No, a person may submit several applications.
- Must *every* such relationship include an person? Yes. The relation would be meaningless without a person.
- Is the result of the relation a function or predicate? Both functions and predicates return a result. *anApplicationOfAPerson* will not return a result.

The properties for **person** are correct as is; click **OK** to close the **Edit role** dialog.

Now repeat the same process for the concept **application** (double-click on **application** in the **Roles** section; the **Edit role** dialog opens). Review the four properties questions:

- The relation is “a person submits an application.” Must *every* application participate in this relation? Yes. Every application must be submitted by a person.
- Does an application participate in *at most one* such relationship? Yes, an application is submitted by only one person.
- Must *every* such relationship include an application? Yes. The relation would be meaningless without an application.
- Is the result of the relation a function or predicate? Both functions and predicates return a result. *anApplicationOfAPerson* will not return a result.

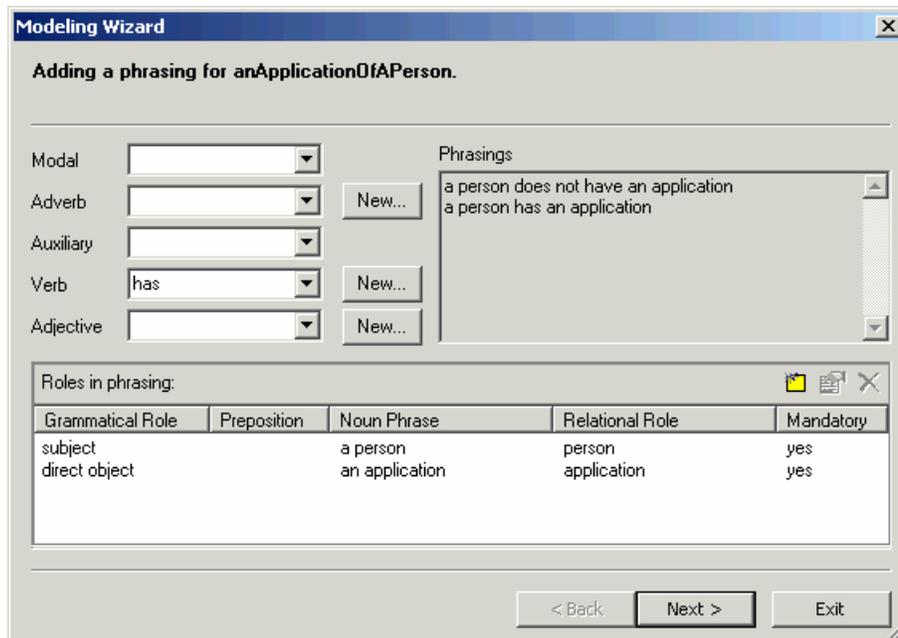
Edit the properties of **application** and then click **OK** to close the **Edit role** dialog.



Click **Next >** to go to the next screen of the modeling wizard. The **Adding a relation between a [concept_name1] and [concept_name2]** screen opens. Select the **add verb phrasing** option and click **Next >**. The **Adding a phrasing for [relation_name]** screen opens.

HaleyAuthority has placed **person** in the role of the subject and **application** in the direct object role (see the **Roles in phrasing** section of the modeling wizard), resulting in the phrasings:

- a person does not have an application
- a person has an application



HaleyAuthority assigns the roles to the concepts according to the order in which you dragged and dropped them. Because you dropped **application** on **person**, HaleyAuthority placed person in the role of the subject and application in the direct object

role. Had you done the reverse, that is, dropped **person** on **application**, HaleyAuthority would have put **application** in the subject role and **person** in the direct object role. This would have resulted in the following phrasing:

- an application does not have a person
- an application has a person

If you inadvertently dropped the concepts in the wrong order, you can easily change the roles of the concepts by double-clicking on the concept in the **Roles in phrasing** section of the dialog to open the **Edit grammatical role** dialog. From this dialog, you can change the role of the concept in the relation.

But we will not need to edit the roles of the concepts for our example, we simply need to change the verb from **has** to **submits**.

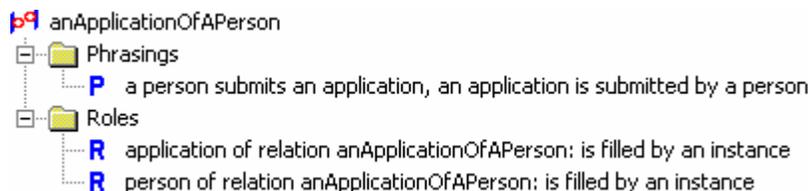
Click on the **Verb** list to display the list of verbs in the HaleyAuthority dictionary – both built-in verbs and any verbs added by users. As you scroll through the list, you will find that **submits**, or any variation of **submit**, is not in HaleyAuthority's dictionary. So you will have to add it.

Click on **New...** next to the **Verb** list. The **Verb** dialog opens. Type *submit* in the **Base form** field. HaleyAuthority will guess the other verb forms and enter them in the appropriate field. Review the other verb forms, as well as the sentences listed in the **Sentences** field, and edit the verb forms as needed. Click **OK** to close the **Verb** dialog.

Submits is entered in the **Verb** field. Check the phrasings; our phrasings are now as follows:

- a person does not submit an application
- a person submits an application
- an application is submitted by a person
- an application is not submitted by a person

Our phrasings are correct, so click **Next >**. The modeling wizard closes and the new relation is listed in the **Relations** node.



Let's check the condition again to see if HaleyAuthority understands it:

a smoker submits the application

HaleyAuthority does understand. Now we are ready to add the next condition.

4.4 A person's age

Let's turn our attention to the condition that limits the applicant's age:

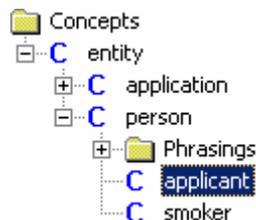
an application should be referred if:

the application's applicant's age is less than 18 years or is at least 40 years

Looking at this statement, can you predict how much HaleyAuthority will understand? One problem that is immediately apparent is that HaleyAuthority doesn't know what an *applicant* is. That situation is easily remedied, but a more significant issue, remains. Does HaleyAuthority understand the relationship between the *applicant* in the condition and the *application* in the statement? If you think about it, we haven't told HaleyAuthority that an application has an applicant, nor have we told it that an applicant has an age. Therefore, when we refer to the applicant's age being a condition for referring the application, we are asking HaleyAuthority act on a relation that has yet to be defined.

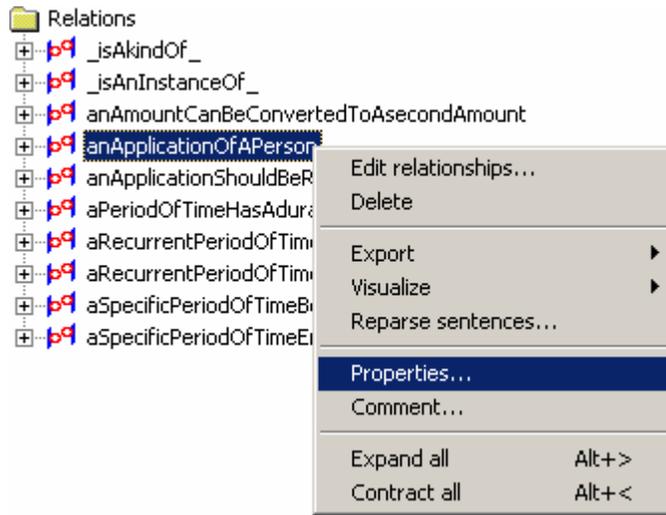
Before we deal with this issue, let's quickly add the entity *applicant*. As was the case when we defined *smoker*, *applicant* is a kind of *person*. Therefore, right-click on **person** and choose **Add a kind of person...** from the menu. The **Adding a phrase** modeling wizard opens. Type in the noun phrase the *applicant* in the **enter noun phrase that identifies the concept** field. Click **Next >**. The **Adding the noun [noun_name]** screen is displayed.

HaleyAuthority has entered default settings for *applicant*. Because an *applicant* is also a *person*, it is appropriate to select the *personal* gender. The rest of the default values pass the sanity check, so click **Next >**. The modeling wizard closes and *applicant* is added to the **entity** node.



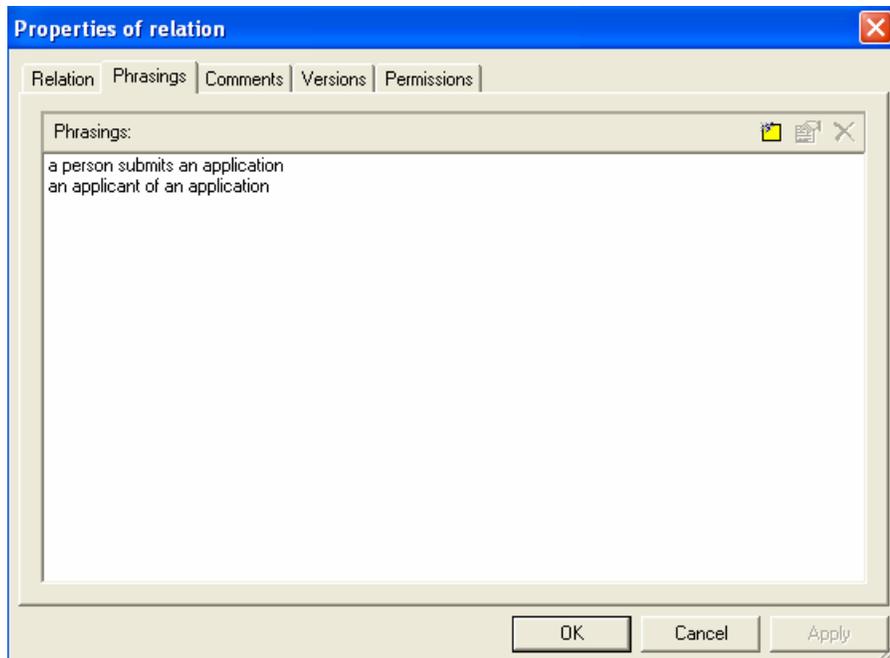
Now let's return to the phrasing of the condition and the relation between **applicant** and **application**. Now that HaleyAuthority knows about applicants and applications, we can tie the two concepts together with a phrasing.

Right-click on the relation *anApplicationOfAPerson* and select **Properties...** from the menu.



The **Properties of relation** dialog opens. Click on the **Phrasings** tab to bring the tab to the front. When we originally created the relation, we gave it the verb phrasing *a person submits an application*, which is listed on the **Phrasings** tab. Now we will add a noun phrasing, *an applicant of an application*.

On the **Phrasings** tab, click . The **Adding a phrasing for [relation_name]** modeling wizard opens. HaleyAuthority has already made a guess at the phrasing and has entered *an application of a person* in the **noun phrasing** field. But we are attempting to create a phrasing that relates applicant and application, so instead of the default phrasing, enter *an applicant of an application* in the field. Click **Next >**. The modeling wizard closes and the new noun phrasing, *an applicant of an application*, is listed in the **Phrasings** tab.



Click **OK** to close the dialog.

To understand what we have done, let's review:

- An *applicant* is a kind of a *person* that submits an *application*.
- We defined the entity *person*.
- We defined the entity *applicant*, which is a specialization of *person*.
- An *applicant* does not have any characteristics of its own, but it inherits characteristics from the generalization *person*.
- We defined the relation a person submits an application.
- We defined the noun phrase *an applicant of an application* for the *person* role of the relation.
- HaleyAuthority now understands that an applicant is a person that submits an application.

Now that we have completed the noun phrasing, we can take advantage of the new phrasing and revise the condition so that HaleyAuthority understands it. Right-click on the statement *an application should be referred* and select **Is applicable if...** from the menu.

The Edit sentence dialog opens. Enter the following condition:

the application's applicant's age is less than 18 years or is at least 40 years

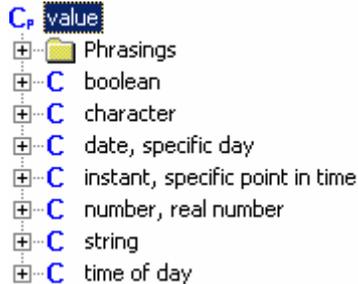
The bold highlighting ends in the middle of the possessive phrase *applicant's age*. HaleyAuthority can't make sense of *applicant's* because it doesn't know that an *applicant* can have attributes. It also doesn't know what an *age* is.

We're going to have to define age, and then let HaleyAuthority know that *a person has an age*. HaleyAuthority will automatically generalize this relation and discover that *an applicant has an age*, too.

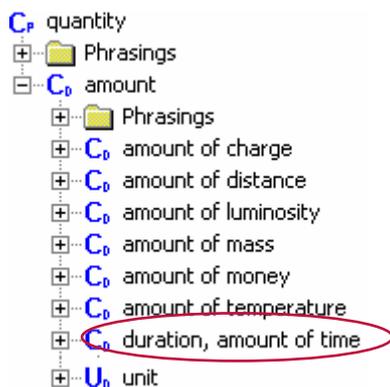
4.4.1 Adding a value

Would *age* be an entity or a value? An entity has characteristics, like height, weight, cost, and color. *Age* is a label on a single numeric value. *Age* is a value.

To define *age* properly, we are going to dive pretty deeply into the **Concepts** tree. First let's look at the kinds of values that are available.



The entries for **date, specific day**, and **time of day** look enticing, but in fact *age* is a **quantity** of time. Expand the **quantity** node and see what is inside it. *Age* is an **amount**; open the **amount** node. The amount node lists **duration, amount of time**, which is the appropriate concept for *age*.

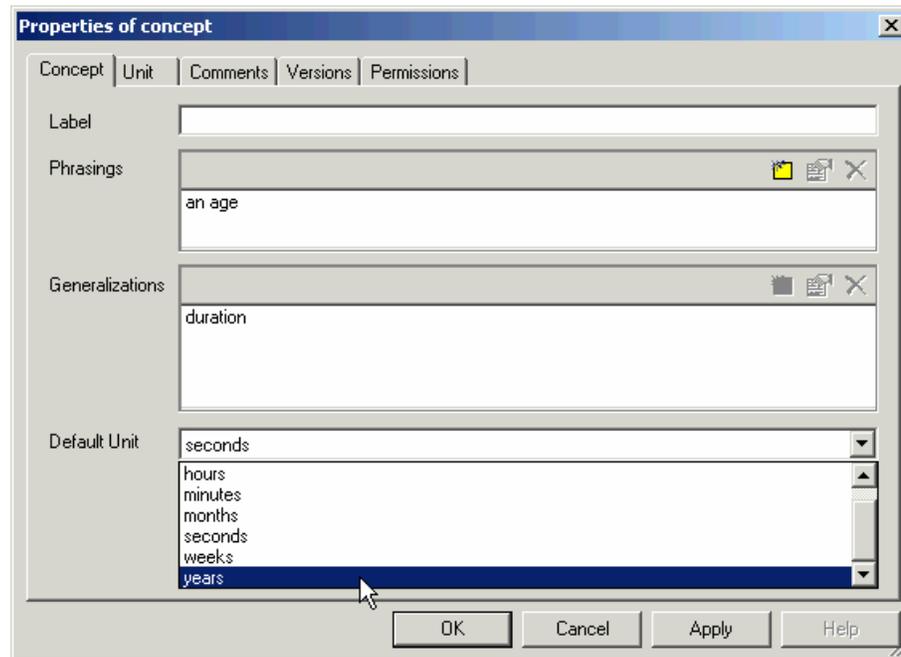


Age is an **amount** of time. Let's add *age* as a **duration, amount of time**. Right-click on **duration, amount of time** and select **Add a kind of duration...** The **Adding a phrase** modeling wizard opens.

Type in *an age* in the **enter a noun phrase that identifies the concept** field and then click **Next >**. The **Adding the noun [noun_name]** screen opens. Check the plural and phrasings. If they make sense, click **OK**. That's all there is to it, with the exception of confirming the units associated with *age*.

What are the different units that can be used to measure time? seconds, minutes, hours, days, weeks, months, and years, just to name a few. But what unit of time do we associate with *age* in particular? We measure *age* in *years*. We need to make sure that *years* is the unit associated with *age*. To do so, we need to look at the **Properties** dialog for *age*.

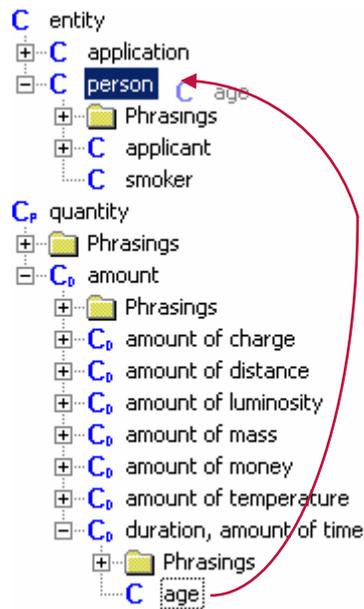
Right-click on *age* and select **Properties...** from the menu. The **Properties of concept** dialog opens, with the **Concept** tab displayed by default. Look at the **Default Unit** field; the default unit associated with *age* is *seconds*. Select *years* from the list and click **OK** to close the dialog.



Now HaleyAuthority will measure *age* in *years*.

4.4.2 *theAgeOfAPerson*

Now that HaleyAuthority knows what an **age** is, we have to explain that a **person** has an **age**. Use the mouse to drag the **age** value up the tree and drop it on the **person** entity. The tree pane will help by scrolling automatically.



Dropping **age** on **person** tells HaleyAuthority that *age* is an attribute of *person*.

HaleyAuthority responds by opening the **Updating a relation's roles and properties** modeling wizard. In the **Roles** section of the screen, double-click on **person** to open the **Edit roles** dialog. Edit the properties of the role *person* to reflect that:

- every *person* must participate in one such relationship
- a *person* participates in at most one such relationship
- every such relationship must have a *person*

Click **OK** to close the **Edit role** dialog and then repeat the same steps to edit the *age* concept. The *age* concept, however, has only one role property: every such relationship must have an *age*. You will note that the default properties for *age* are correct, so you can close the **Edit roles** dialog. Click **Next >** on the modeling wizard screen. The **Adding a relation** screen opens.

Enter the noun phrase, *the age of a person*, in the **add noun phrasing** field and then click **Next >**. The modeling wizard closes and the relation, *APersonHasAnAge*, is added to the **Relations & Procedures** node.

HaleyAuthority knows what an *age* is, and it knows that *a person has an age*. Let's go check that incomplete statement about the applicant's age.

When we open the **Edit condition** dialog, we can see at a glance that we are done.

the application's applicant's age is less than 18 years or is at least 40 years

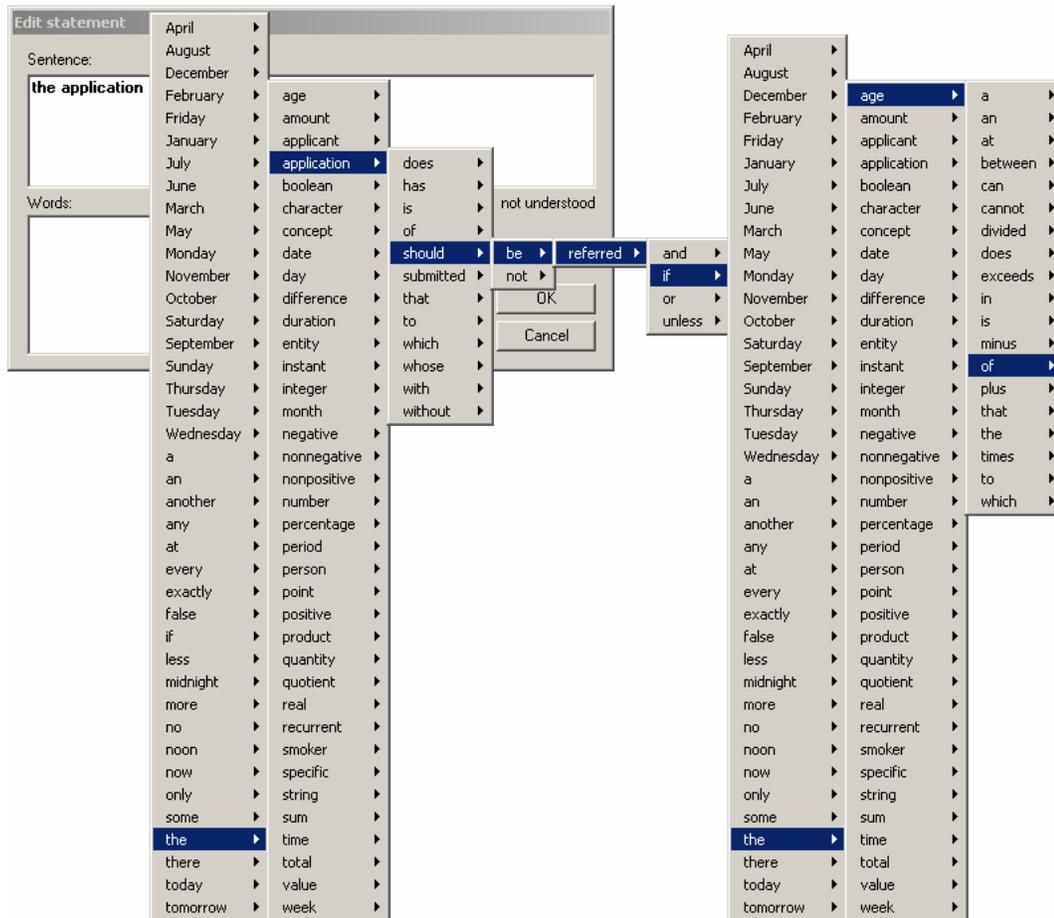
HaleyAuthority understands the complete statement. Click **OK** to save the condition as understood and to close the dialog.

4.4.3 Creating a point-and-click statement

We're going to digress for a moment to show you one of HaleyAuthority's most impressive features. When you create a statement, HaleyAuthority actually *predicts* what you are about to type next. You can access these predictions as cascading menus in the **Edit statement** and **Edit condition** dialogs.

Open the **Edit statement** dialog, but don't type anything into it. Instead, right-click the mouse anywhere in the dialog. This opens a waterfall of cascading menus. You can roll the mouse along the menus and build any statement that HaleyAuthority knows how to interpret.

For example, the cascading menus can take you quite far without typing a keystroke:



When HaleyAuthority is unable to predict the next correct word in the sentence, click on the final correct word on the menus; HaleyAuthority closes the cascading menus and enters the portion of the statement that you selected from the cascading menus into the

Sentence field. Click in the **Sentence** field and type any additional words that are necessary to complete the sentence.

The cascading menus are efficient, but that is not their real value. They show you all of the words and phrases HaleyAuthority knows how to interpret that *you didn't have to create yourself*. You can also see these options in the **Words** field in the lower left corner of the **Edit statement and Edit condition dialogs**. HaleyAuthority has a great deal of built-in knowledge. That is where its true power lies.

4.5 Hazardous occupations

The next condition to add to the statement *an application should be referred if is:*

the occupation of the person who submits the application is hazardous

After you add the condition, you will need to do the following so that HaleyAuthority can understand the condition:

- Add the entity *occupation*
- Define a relation, an occupation is hazardous
- Add instances of occupations (both hazardous and non-hazardous)
- Add the fact that an occupation is hazardous to those *occupation* instances that are determined to be hazardous
- Define a relation that links applicant to occupation – an occupation of a person

4.5.1 Adding the entity *occupation*

There might be many possible types of occupations, and each might have various attributes. Clearly, an occupation is not simply a value, so it must be an entity. Right-click on the **entity** node, and select **Add a kind of an entity...** This opens the **Adding a phrasing** modeling wizard. Type the noun phrasing of the entity, *an occupation*, in the **enter a noun phrase that identifies the concept** field. Click **Next >**. The **Adding the noun [noun_name]** screen opens.

Verify that the plural form of *occupation* is correct (*occupations*), the attributes are correct (*count* noun), the gender is correct (*neuter*), and the phrasings are correct (*1 occupation*, *2 occupations...*). The default settings should be correct. Click **Next >**. The modeling wizard closes and HaleyAuthority adds *occupation* to the **entity** node.

4.5.2 Defining the relation an occupation is hazardous

Now we will define the relation, an occupation is hazardous. This relation will allow us to signify which of the occupation instances (which we will add next) are hazardous.

Right-click on **occupation** and select **Add a relation involving an occupation...** from the menu. The **Updating the relation's roles and properties** modeling wizard opens. We will call the relation *anOccupationIsHazardous*; type that name in the **Name** field. If you look at the **Roles** section of the wizard, you see that there is only one role in this relation – *occupation*. Let's take a look at the properties of the concept. Double-click on *occupation* or select *occupation* and click  to open the **Edit role** dialog.

The default property for *occupation* within the relation *anOccupationIsHazardous* is that **every such relation must have an occupation**. That makes sense, doesn't it? As we discussed earlier, *occupation* is the only concept in the relation, therefore the relation obviously requires the inclusion of *occupation*.

We also want to choose another property option for the relation, as well. Check the **an occupation participates in at most one such relationship** option. Therefore, *occupation* is **unique**. This means that if an occupation is hazardous, it is always hazardous. For example, if the instance of an occupation is *snake charmer*, and *snake charmer* has the phrasing, an occupation is hazardous, then any applicant that has the occupation of snake charmer will be considered by HaleyAuthority to be a person with a hazardous occupation.

Click **OK** to close the **Edit role** dialog and to return to the modeling wizard. Click **Next >**. The **Adding a relation...** screen opens. We want to add the verb phrasing *an occupation is hazardous*, so select the **add verb phrasing** option and click **Next >**. The **Adding a phrasing...** screen opens.

If you look in the Phrasings section, you see the default phrasings are:

- an occupation has
- an occupation does not have

The first thing we need to do is to change the verb from *has* to *is*. Select *is* from the **Verb** list and then check the phrasings again. Now the phrasings are:

- an occupation is
- an occupation is not

Now all we need to do is to add *hazardous* to the phrasing and we will be done.

What is *hazardous*? It is an adjective that modifies *occupation* – a *hazardous occupation*. Look at the **Adjective** list and see if *hazardous* is already in HaleyAuthority’s dictionary. No, it is not, so we can add the word from within this modeling wizard by clicking **New...** next to the **Adjective** list. The **Adjective** dialog opens.

Type *hazardous* in the **Adjective** field. Now we need to answer some questions about the adjective:

- Is *hazardous* attributive? Can you use it in front of a noun? Yes, you can – a *hazardous occupation*. Therefore, you should place a checkmark in the **Attributive** option.
- Is *hazardous* gradable? Can you precede it with *very*? Yes, something can be *very hazardous*. Therefore, you should place a checkmark in the **Gradable** option.

Because you selected the **Gradable** option, HaleyAuthority makes a guess as to what *hazardous* would be in the comparative (*-er*) form: *hazardouser*. That is obviously incorrect; *hazardous* does not have a comparative form (instead, we would speak of something being *more hazardous*), so delete *hazardouser* from the **Comparative form** field.

HaleyAuthority also makes a guess as to what *hazardous* would be in the superlative (*-est*) form: *hazardousest*. Again, that is obviously incorrect; *hazardous* does not have a superlative form (instead, we would speak of something being *most hazardous*), so delete *hazardousest* from the **Superlative form** field. Click **OK** to save the new adjective and close the **Adjective** dialog. Back in the modeling wizard, we see that *hazardous* is now the selected adjective.

Let’s check the phrasings one last time:

- an occupation is hazardous
- an occupation is not hazardous

Our phrasings are correct. Click **Next >**. The modeling wizard closes, and **occupation** now has the new phrasing: *an occupation is hazardous*.

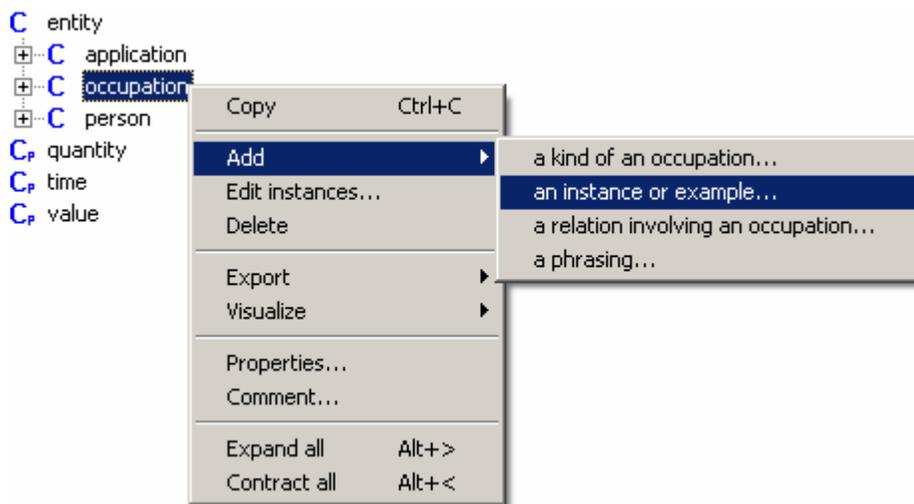


4.5.3 Adding instances of an *occupation*

How do we know that an applicant has a hazardous occupation? In a real application, we might know this in a dozen ways, but let's just give HaleyAuthority some examples of hazardous and non-hazardous occupations to use as a guide.

The first step is to create a few occupations, hazardous and otherwise. We do this by defining *instances* of the appropriate entities.

An instance is a specific example of an entity; *firefighter* is a specific example from the class of occupations. To create these instances, right-click *occupation* and select **Add an instance... from the menu.**



The **Instance** dialog opens. Type the name of the instance (in this case, type *firefighter*, in the **Label** field and then click **OK** to close the dialog. The instance is added to the entity in the tree, preceded with the **I** icon, which indicates an instance. You will add the following occupations: firefighter, lawyer, pharmacist, scuba diver, snake charmer, and used car salesman.

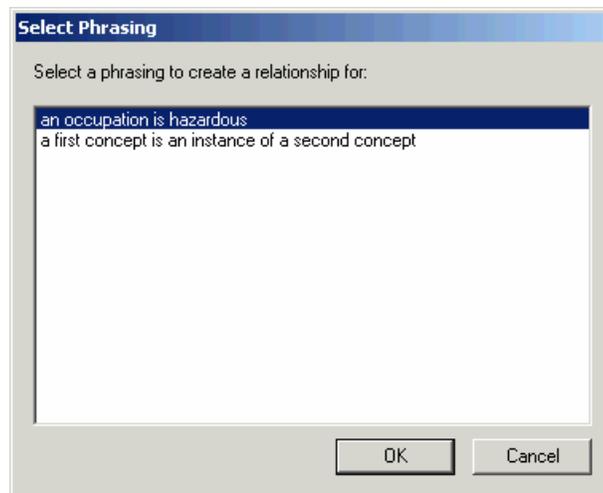


4.5.4 Adding a fact to an *occupation* instance

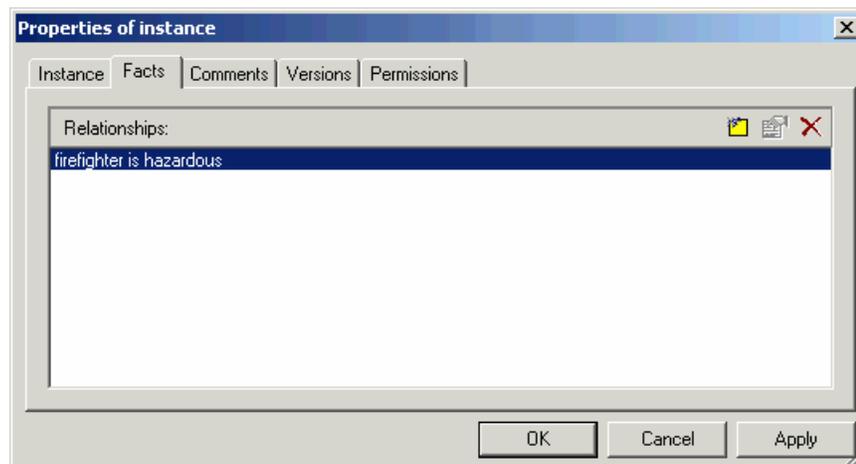
How will HaleyAuthority know that an occupation is hazardous? We will tell HaleyAuthority by adding a fact, *an occupation is hazardous*, to those occupations deemed hazardous.

Right-click on an instance of an occupation that is hazardous. In our example, hazardous occupations are *firefighter*, *scuba diver*, and *snake charmer*. Select **Properties...** from the menu. The **Properties of instance** dialog opens, with the **Instance** tab displayed by default. Click on the **Facts** tab to bring it to the front of the dialog.

We want to add a fact, so click . The **Select Phrasing** dialog opens, it shows two facts they you can assign to the instance:



Select the phrasing *an occupation is hazardous* and then click **OK** to close the dialog. The new phrasing is listed in the **Instance** tab.



Click **OK** to close the dialog. The new fact is listed in the tree:



Now add the *an occupation is hazardous* fact to the remaining hazardous occupations.

4.5.5 Adding the phrasing *an occupation of a person*

HaleyAuthority understands about *persons* and *applicants*. It understands about *occupations* and *hazardous occupations*. The next step is to tell HaleyAuthority about the *occupation of a person*. Once HaleyAuthority understands that a person has an occupation, it will automatically generalize to all types of persons, and to all types of occupations.

How difficult is it to tell HaleyAuthority about *an occupation of a person*? All you do is drag **occupation** down the tree and drop it on **person**. The **Updating the relation's roles and properties** modeling wizard opens. Type the name of the new relation, *anOccupationOfAPerson*, in the **Name** dialog. The default properties of the roles, *occupation* and *person*, do not need to be edited, so click **Next >**.

The **Adding a relation...** screen opens, with the phrasing *an occupation of a person* entered in the **add noun phrasing** field. Accept the noun phrasing and then click **Next >**. The modeling wizard closes and the noun phrasing is added to the tree.

Now HaleyAuthority knows that a person can have an occupation, and it also knows that *an applicant* can have a *hazardous* occupation.

Let's check the condition and see if HaleyAuthority now understands everything it needs to know about hazardous occupations:

the occupation of the person who submits the application is hazardous

Yes, HaleyAuthority does understand hazardous occupations. Notice how HaleyAuthority generalizes behind the scenes. We told it *a person has an occupation*. We told it that *an applicant is a person*. We defined *occupation* and appended a fact to hazardous occupations that told HaleyAuthority that particular occupations are hazardous.

At this point we discover that HaleyAuthority is completely comfortable with the phrase, *the occupation of the person who submits the application is hazardous*. It knows *exactly* what we mean.

4.6 The applicant's coverage and income

The next condition states that an application should be referred if *the application requests coverage for more than 80% of the income of the application's applicant*.

First, we will type the statement into the **Edit condition** dialog to see how much HaleyAuthority understands:

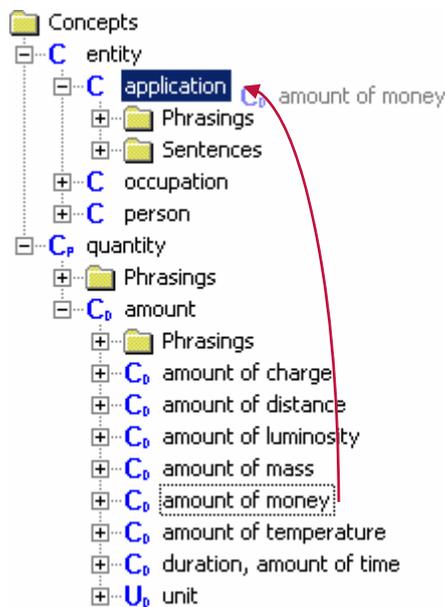
the application requests coverage for more than 80% of the income of the application's applicant

It doesn't appear to understand too much at this point, but that is easily remedied. We will:

- Add the relation *anApplicationRequestsAnAmountOfMoneyInCoverage* that relates the concepts amount of money (because coverage is an amount of money) and an application
- Add the relation *aPersonEarnsAnAmountOfMoney* that relates the concepts *amount of money* (because *income* is an *amount of money*) and a *person*.
- Add the noun phrasing the requested coverage of an application

4.6.1 Adding the relation *an application requests coverage for an amount of money*

We speak of *coverage* as the name of a single number representing an amount of money. Navigate into the **quantity** tree, looking for an amount expressed as *an amount of money*. When you find it, drag it and drop it on the entity **application**.



The **Updating the relation's roles and properties** modeling wizard opens. Type the relation name in the **Name** field, such as, *anApplicationRequestsAnAmountOfMoneyInCoverage*.

We need to change some properties for *application*, so in the **Roles** section of the screen, double-click on *application* to open the **Edit role** dialog. By default, HaleyAuthority has checked the option **every such relationship must have an application**. This makes sense, because it is in the application that the applicant requests an amount of coverage. We also want to check the **every application must participate in a least one such relationship** option, because we are requiring that every application include a request for an amount of coverage. Finally, we want to check the option **an application participates in at most one such relationship**. There will be only one application per request for an amount of coverage. Any additional request for an amount of coverage will require another application. Click **OK** to close the dialog and return to the modeling wizard.

The **Adding a relation** screen opens. We want to add a verb phrasing, so select the **add verb phrasing** option and click **Next >**. The **Adding a phrasing** screen opens. Let's look at the default phrasing first:

- an application does not have an amount of money
- an application has an amount of money

The first thing you probably noticed is that the concept *coverage* is absent from the phrasing. We are building a ternary (three-part) relation – *application*, *amount of money*, and *coverage*. Let's review the desired phrasing again: *an application requests coverage for an amount of money*. In this phrasing, we have three grammatical roles. *Application* is the subject, *coverage* is the direct object, and *for an amount of money* is a prepositional phrase.

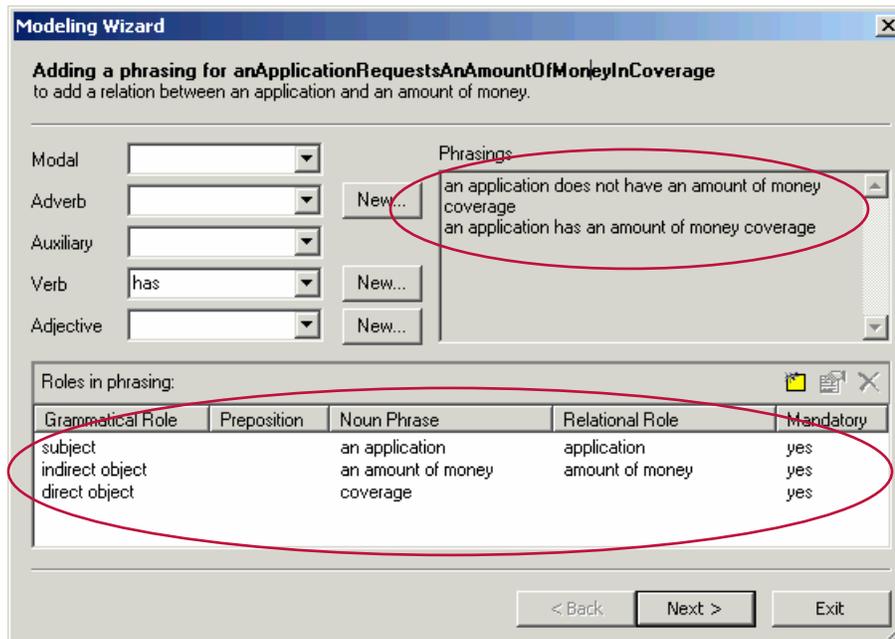
Without the concept *coverage* in the relation, HaleyAuthority has guessed correctly that *application* is the subject, but it has guessed incorrectly that *amount of money* is the direct object. To fix this, we will add the concept *coverage* and put it in the direct object role. In the **Roles in phrasing** section of the screen, click . The **Edit grammatical role** dialog opens.

We have already determined that *coverage* is the direct object, so select *direct object* from the **Grammatical Role** list. Change the **Type of Role** to *Syntactic text*, because while adding *coverage* to the phrasing will make the phrasing more graceful, *coverage* doesn't actually participate in the underlying relation that HaleyAuthority will convert to an Eclipse relation. Eclipse merely needs to know that an application requests an amount of

money. The fact that we call that amount of money *coverage* is immaterial. Finally, type *coverage* in the **Text** field. Click **OK** to close the dialog.

A **Yes or No** dialog opens. HaleyAuthority warns you that another concept is currently in the direct object role. As we discussed earlier, HaleyAuthority had placed *amount of money* in the direct object role by default. You are asked if you want to “unassign” the direct object role so that it can be “reassigned” to *coverage*. Only one concept can hold the role of direct object, and we know that it should be *coverage*, so click **Yes**.

We return to the modeling wizard, which displays the updated roles in the **Roles in phrasing** section.



Coverage is now in the direct object role, and HaleyAuthority has changed *amount of money* from a direct object to an indirect object. We know that isn't correct, we already determined that *amount of money* is part of the prepositional phrase *for an amount of money*. In a few minutes, we will assign *amount of money* to the role of prepositional phrase that *follows coverage*, but first, let's look at the current status of the phrasings:

- an application does not have an amount of money coverage
- an application has an amount of money coverage

Not only do we need to change the grammatical role of *amount of coverage*, we also have to change the verb from *has* to *requests*.

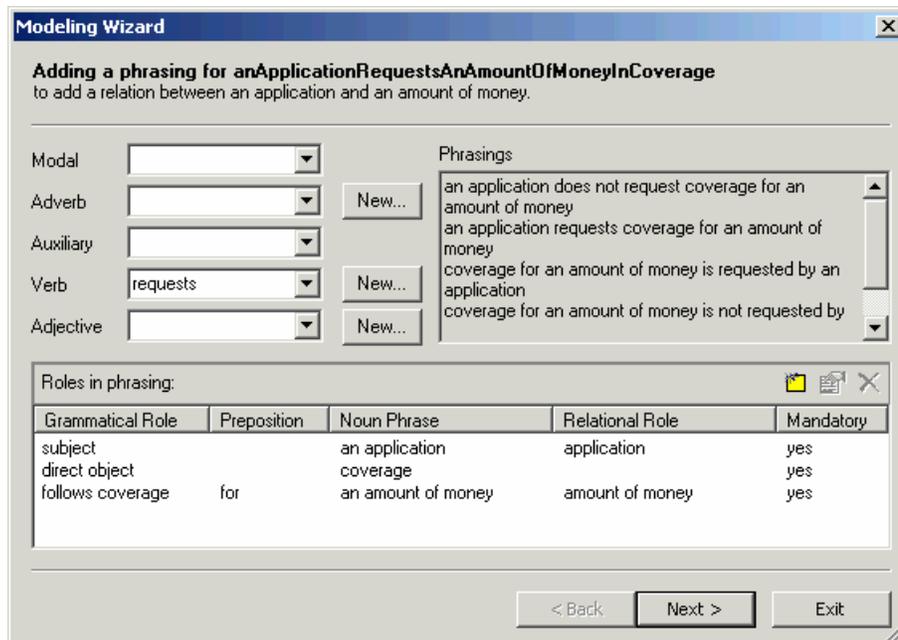
Click on the **Verb** list to see if *requests* is in HaleyAuthority's dictionary. It is not, so click **New...** next to the **Verb** list to open the **Verb** dialog. Enter the base form, *request*, in the

Base field. HaleyAuthority will fill in the other forms of the verb. Review all of the forms, as well as the sentences, to make sure that HaleyAuthority entered the correct forms. Click **OK** to close the dialog and return to the modeling wizard. Requests is automatically selected in the Verb list, and the phrasings have changed to the following:

- an application does not request an amount of money coverage
- an application requests an amount of money coverage

Now we simply need to fix the grammatical role of amount of money and the phrasings will be correct. In the **Role in phrasings** section of the screen, double-click on *amount of money* to open the **Edit role** dialog. We know that *amount of money* will follow *coverage* in the phrasings, so select *follows coverage* from the **Grammatical Role** list.

HaleyAuthority has correctly guessed that **Type of role** is *Relational*. The phrasing is *for an amount of money*, so select *for* from the **Preposition** list. Finally, HaleyAuthority has correctly selected *an amount of money* from the **Relational Role** list. The settings for this role are complete, so click **OK** to close the dialog and return to the modeling wizard.



We now have the phrasings we want, so click **Next >**. The modeling wizard closes.

Let's check HaleyAuthority's progress on understanding the condition:

the application requests coverage for more than 80% of the income of the application's applicant

4.6.2 Adding the phrasing *an income of a person*

Income, like *coverage*, is a quantity that measures an *amount of money*. In the noun phrase we are about to add, we want to relate the concepts *income* and *person*. To add the phrasing, drag *amount of money* up the tree and drop it on *person*.

The **Updating the relation's roles and properties** modeling wizard opens. Type the relation name in the **Name** field, such as, *aPersonEarnsAnAmountOfMoney*. The default properties of the roles are correct; there is no need to edit the roles. Click **Next >**.

The **Adding a relation** screen opens. We want to add a noun phrasing, so type *an income of a person* in the **add noun phrasing** option and click **Next >**. Because HaleyAuthority does not understand *income*, the **Adding the noun...** screen opens. The default settings for *income* are correct, so click **Next >**. The modeling wizard closes.

Let's check HaleyAuthority's progress on understanding the condition:

**the application requests coverage for more than 80% of the income of the
applicant's applicant**

The condition is now understood.

Next we will move on to adding a series of health questions and telling HaleyAuthority what to do with the applicant's answers.

4.7 The applicant's answers

The applicant is asked to answer three questions pertaining to his or her health. If the applicant answers *yes* to any of the questions, then the application should be referred. We need to add a condition that will evaluate the answers to the questions. Therefore, add the following condition to the statement, *an application should be referred*, if;

a health question on the application was not answered with *No*

HaleyAuthority doesn't know about questions and answers yet. We're going to have to explain in some depth. We'll have to tell HaleyAuthority what a question is. A question could have properties, such as the text of the question and the correct answer, so a question is probably an entity. There might be multiple types of questions, so we probably should define a "health question" as a subentity of question. There will be specific instances of health questions, such as Question 1, Question 2, and Question 3. This much is clear.

We can easily classify answers by specifying two instances: *Yes* and *No*.

Finally, we want to define the following relation that connects the entities *question*, *application*, and *answer*: *a question on an application was answered with an answer*.

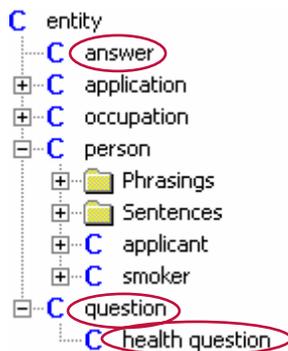
In review, so that HaleyAuthority can understand the condition, we will:

- Add the entities *question*, *health question*, and *answer*
- Add *Yes* and *No* as instances of *answer*
- Add the relation a question on an application was answered with an answer

4.7.1 Adding the entities *question*, *health question*, and *answer*

We have already defined **applicant**, but *questions*, *health questions*, and *answers* are new entities. We need to create an **answer** entity, a **question** entity, a **health question** subentity. **Health question** can be added as a compound noun. By now this should be easy, so we'll leave the details to you.

When you are finished, you should see the following:



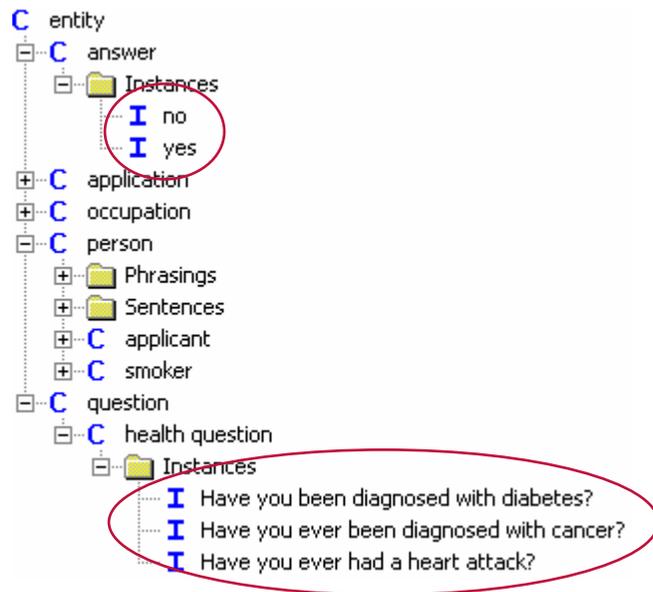
4.7.2 Adding instances of a *health question* and instances of an *answer*

Now add the instances of *health question*:

- Have you been diagnosed with diabetes?
- Have you ever been diagnosed with cancer?
- Have you ever had a heart attack?

and the instances of *answer*:

- no
- yes

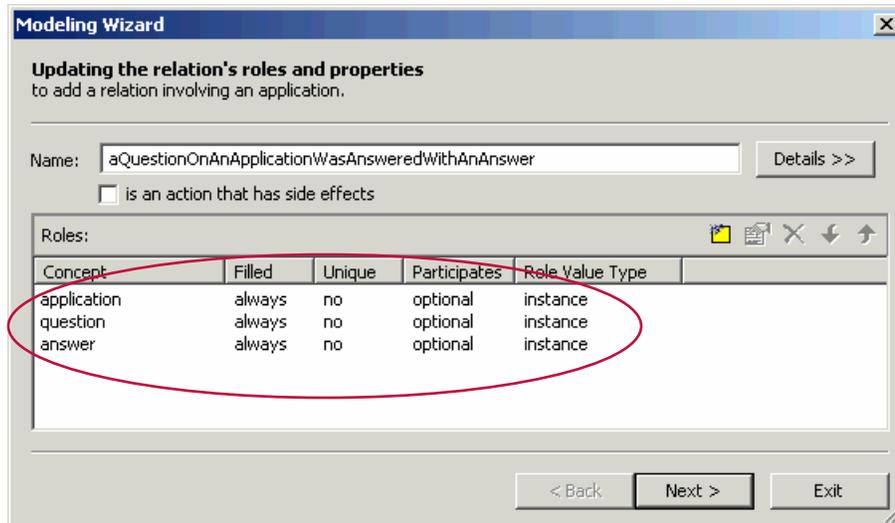


4.7.3 Adding the relation *aQuestionOnAnApplicationWasAnsweredWithAnAnswer*

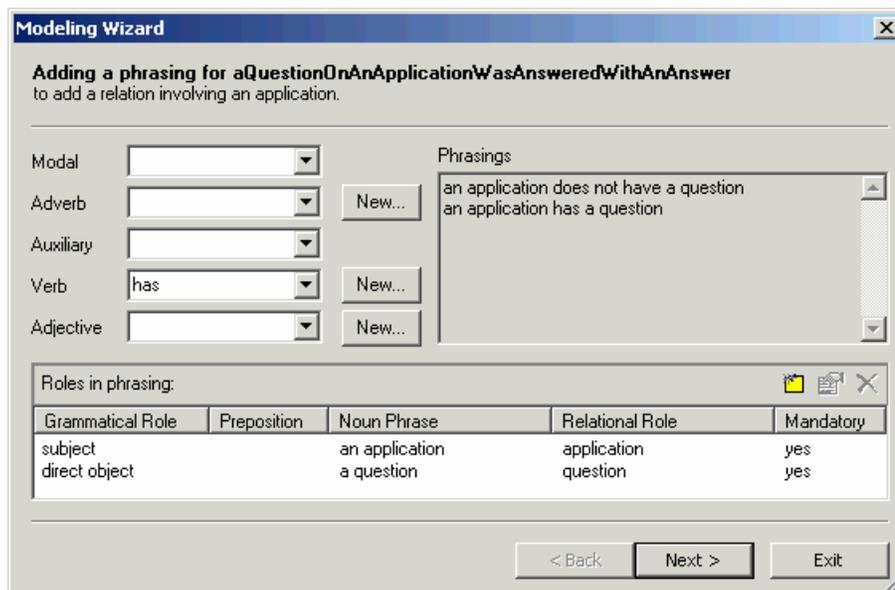
Now we can build the relation. We will not use the drag-and-drop technique this time because that works best for *binary* (two-part) relations, and this relation has three parts (*application*, *question*, and *answer*).

Right-click the **application** entity and select **Add a relation involving an application...** from the menu. This opens the **Updating the relation's roles and properties** modeling wizard. Type the relation name, *aQuestionOnAnApplicationWasAnsweredWithAnAnswer*, in the **Name** field.

Click  in the **Roles** section of the modeling wizard to add the concept *question*, and then again to add the concept *answer*. We now have all three parts of the relation. Happily, the default properties for all three concepts are correct, so it is not necessary to open the **Edit role** dialog to edit the properties.



Click **Next >**. The **Adding a relation involving an application** screen opens. We want to create a verb phrasing, so select the **add verb phrasing** option and click **Next >**. The **Updating the relation's roles and properties** screen opens, but it shows only the *application* and *question* concepts, not the *answer* concept. Why not?



When you create relations with three or more elements, HaleyAuthority gets cautious about assigning default syntactic roles in phrasings. It assumes that the first concept is the subject (in this case, *application*) and the second concept is the direct object (in this case, *question*), but then it leaves the others concepts to you.

Before we add the *answer* concept, however, let's figure out the grammatical roles of *application* and *question*, as well as add any modals, adverbs, auxiliaries, verbs, or adjectives that will be required to build the relation.

Compare the current phrasing to the desired phrasing:

- **Current phrasing:** an application has a question
- **Desired phrasing:** a question on an application was answered with an answer

The first thing we need to do is change the subject from *application* to *question*. Double-click *question* to open the **Edit grammatical role** dialog and change the **Grammatical role** to *subject*. Click **OK** to close the dialog. HaleyAuthority warns you that another concept already holds that role (*application*) and asks if you want to unassign the role. Click **Yes**.

Let's check the phrasing again:

- **Current phrasing:** a question has an application
- **Desired phrasing:** a question on an application was answered with an answer

We need to change the relation between *question* and *application* so that it *follows question* and is joined with the preposition *on*. Use the **Edit grammatical role** dialog to change the **Grammatical Role** of *application* to *follows a question* and then add the preposition *on*.

Check the phrasing again:

- **Current phrasing:** a question on an application has
- **Desired phrasing:** a question on an application was answered with an answer

We are making quick progress in building this relation. Next we will change the verb phrase from *has* to *was answered*.

Select *was* from the **Auxiliary** list, and then look for **answered** in the **Verb** list. You might expect to find *answer* in the list because you just added the concept *answer* to build this relation. But you added *answer* as a noun, and we need to add *answer* as a verb.

Click the **New...** button next to the **Verb** list to enter the **Verb** dialog. Type *answer* in the **Base** field; HaleyAuthority will guess at the various forms of the verb. While HaleyAuthority does a good job guessing the plural (**-s**) form, it has a bit of trouble with the spelling of the **-ing participle**, **past**, and **-ed participle** forms. Correct the spelling of these forms of the verb *answer* (it should be, respectively, *answering*, *answered*, and *answered*) and then close the dialog. HaleyAuthority has automatically selected *answering* from the **Verb** list. That is not quite right, though – we want *answered*. Select *answered* from the **Verb** list.

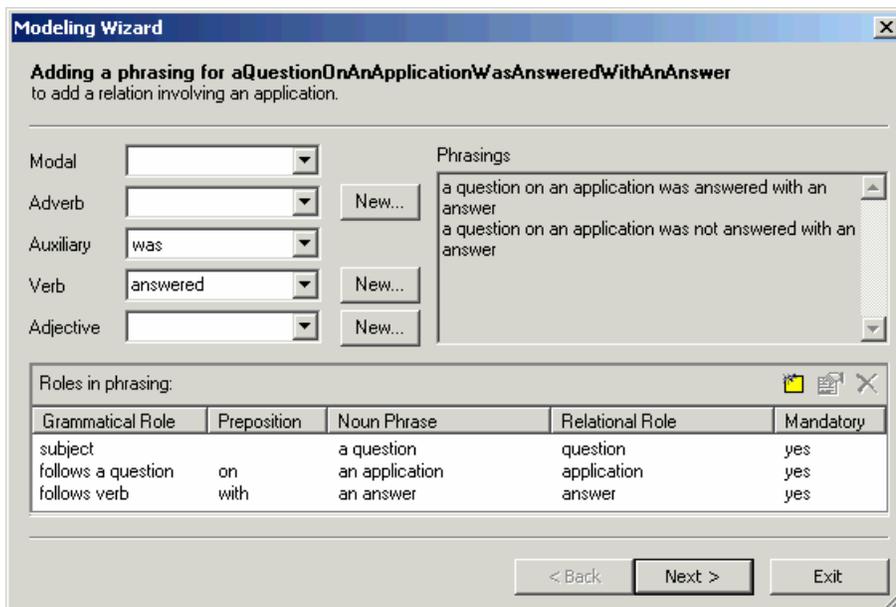
How is the phrasing coming?

- **Current phrasing:** a question on an application was answered
- **Desired phrasing:** a question on an application was answered with an answer

Now we are ready to add the third concept to the relation.

Click  to open the **Edit grammatical role** dialog. We know that the position of *answer* in the relation is following the verb. Therefore, select *follows verb* from the **Grammatical Role** list. Because *answer* is part of the prepositional phrase *with an answer*, select *with* from the **Preposition** list. Finally, we select the concept noun phrasing from the **Relational Role** list; select *an answer*. Click **OK** to close the dialog.

Take a final look at our phrasing:



Grammatical Role	Preposition	Noun Phrase	Relational Role	Mandatory
subject		a question	question	yes
follows a question	on	an application	application	yes
follows verb	with	an answer	answer	yes

Our phrasing is correct. Click **Next >**. The modeling wizard closes. Now, let's check how much of the condition HaleyAuthority understands.

a health question on the application was not answered with No

HaleyAuthority understands the condition perfectly.

4.8 Height and weight limits

The KBA should take a hard look at any policy that refers to a table. HaleyAuthority can deal with table-based policies, but first do a sanity check. Where did the table come from? If the table was generated by a formula in a spreadsheet, it is easier to allow HaleyAuthority to use the same formula rather than input the table values and then teach HaleyAuthority how to interpolate values.

The raw policy said, *if the build table height and weight are within normal limits then the applicant might be eligible*. A little research reveals that the table is based on a *body mass index* formula. *Normal* was defined as a body mass index between 20 and 30. That creates a new policy: An application should be referred if *the applicant's body mass index is less than 20 or is more than 30*.

The body mass index (BMI) is calculated as follows:

$$\text{BMI} = \text{Weight} * 703 / \text{Height}^2$$

An applicant's body mass index is the applicant's weight in pounds times 703 divided by the square of the applicant's height in inches. (We are specifying *pounds* and *inches* in this example to show you HaleyAuthority's ability to convert units automatically, something that we will discuss later in this example.)

We need to enter two statements and a condition. For now, we will enter the statements in the **Referral Module** (we will later create a new module, **Calculation Module**, and move these statements to the new module). The condition will be entered along with the other conditions we have entered to this point. The statements look like this:

a person's body mass index is the person's weight in pounds times 703 divided by the square of the person's height in inches

the square of a number is the number times the number

And the condition on the "an application should be referred" statement looks like this:

the body mass index of the application's applicant is not between 20 and 30

HaleyAuthority does not seem to know about *body mass index*, *height*, *weight*, or *square of the height*. We will have to tell it about all of these values, and associate them with the *applicant* entity.

Here is a related point. HaleyAuthority understands that numeric values often have units, and it understands that numeric values can also have compound units. Therefore, if you write a rule that states that two different units, such as *pounds* multiplied by *square foot*, equals a third unit, *pounds per square foot*, you can then implement that rule in other statements or conditions.

For the purposes of this example, however, we do not need to use compound units, because *body mass index* has no units (despite the fact that it is calculated by using two different units – *pounds* and *inches*). An *index* has no units; it is simply a floating-point number we compare with a threshold. We would define an applicant's BMI as a simple float.

To allow HaleyAuthority to understand the two statements and condition that allow us to calculate BMI, we will:

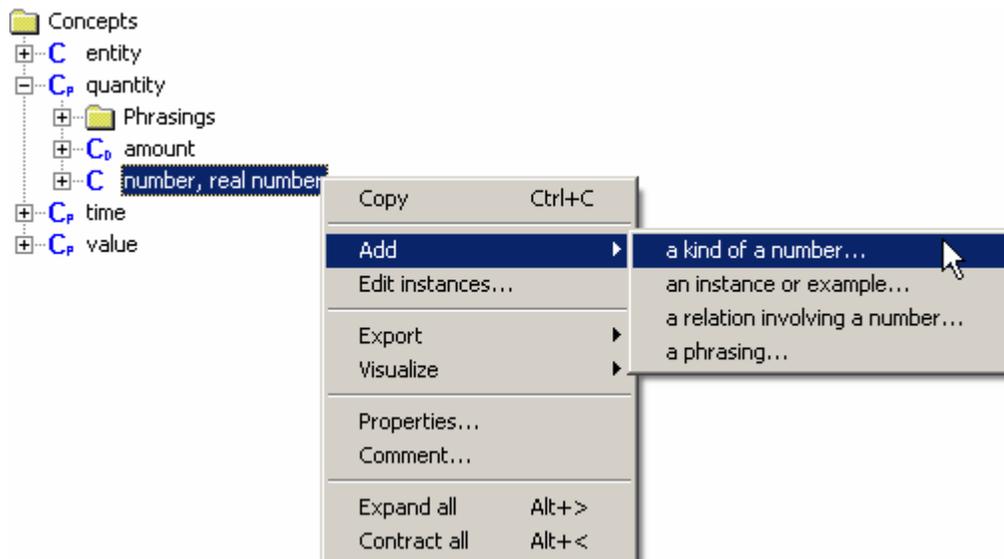
- Add the concepts:
 - *body mass index* and *square* (number)
 - *height* and *weight* (amount)
- Add the noun phrasings the body mass index of a person, the weight of a person, the height of a person and the square of a number

4.8.1 Adding the concepts *body mass index*, *square*, *height*, and *weight*

Both *body mass index* and *square* are *numbers* (**Concepts > quantity > number, real number**). *Height* and *weight*, however, are *amounts* (**Concepts > quantity > amount**).

4.8.1.1 *Body mass index* and *square*

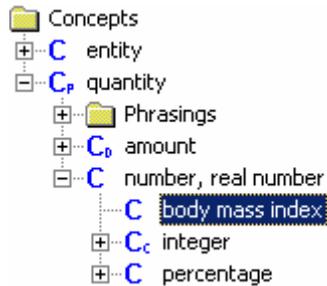
First, we will create *body mass index*. Navigate into the **quantity** node and look for unit-free numeric values. For the most accurate results, we would prefer to be dealing with *real numbers* (floats). Right-click on **number, real number** and select **Add a kind of a real number...** from the menu.



The **Add a phrase** modeling wizard opens. Type a *body mass index* in the **enter a noun phrase that identifies the concept** field. Click **Next >**. The **Identifying compound noun** screen of the modeling wizard opens. The wizard prompts you to identify any compound nouns in the noun phrase.

Check the **body mass index** checkbox to indicate that *body mass index* is a compound noun and then click **Next >**. The **Adding a compound noun...** screen opens.

HaleyAuthority knows that the noun *body mass index* is not in the knowledgebase dictionary, so the wizard prompts you to add *body mass index* as a new noun. Accept the default properties defined by HaleyAuthority for *body mass index* by clicking **Next >**. The modeling wizard closes and the new value, *body mass index*, is added to the knowledge tree.



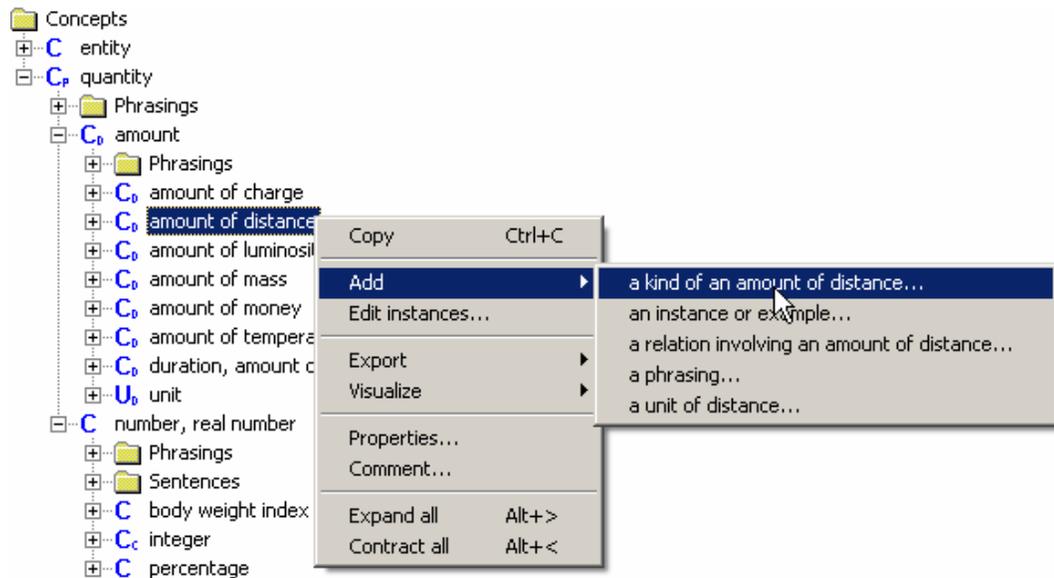
Next we will create the value *square*.

As in the case of *body mass index*, there is no necessity to associate a unit with the *square* value. Therefore, create the value *square* in the same manner as *body mass index*.

4.8.1.2 *Height and weight*

Now we will create the value *height*; *height* is an *amount of distance*.

Right-click on *amount of distance* (**Concepts > quantity > amount > amount of distance**) to open its **Object** menu. Select **Add > a kind of an amount of distance...** from the **Object** menu.



The **Adding a phrase** modeling wizard opens.

Enter the noun phrase, *a height*, in the field beneath the **enter a noun phrase that identifies the concept** option and click **Next >**. The **Adding a noun...** screen is displayed.

HaleyAuthority doesn't understand the noun *height*, so the modeling wizard prompts you to add *height* as a new noun. Accept the default properties defined by HaleyAuthority for *height* by clicking **Next >**. The concept's properties window is now displayed. Notice that the default unit for the height concept is already selected as inches. Accept the defaults and click **Next >**. The modeling wizard closes and *height* is added to the knowledge tree.

Next, we will create the value *weight*, *weight* is an *amount of mass*.

Right-click on *amount of mass* (**Concepts > quantity > amount > amount of mass**) and select **Add > a kind of an amount of mass...** from the menu. The **Adding a phrase** modeling wizard opens. Enter the noun phrase, *a weight*, in the field beneath the **enter a noun phrase that identifies the concept** option and click **Next >**. The **Adding a noun...** screen is displayed.

HaleyAuthority doesn't understand the noun *weight*, so the wizard prompts you to add *weight* as a new noun. Weight is both a *count* noun and a *mass* noun. Check the **mass** checkbox and then accept the remainder of the default properties defined by HaleyAuthority for *weight* by clicking **Next >**. The concept's properties window is now displayed. Notice that the default unit for the weight concept has defaulted to ounces, which is incorrect. Change this value to pounds and click **Next >**. The wizard closes and the new value, *weight*, is added to the knowledge tree.

We have added all four of the concepts we require for the two statements and conditions; check to see if HaleyAuthority understands the statements:

a person's body mass index is the person's weight in pounds times 703 divided by the square of the person's height in inches

the square of a number is the number times the number

the body mass index of the application's applicant is not between 20 and 30

Very little has changed! Why not? It is not enough to create the values; the values are meaningless until they appear in relations.

4.8.2 Adding the relations *theHeightOfAPerson*, *theWeightOfAPerson*, *theBodyMassIndexOfAPerson*, and *theSquareOfANumber*

The next step is to add the relations, *theHeightOfAPerson*, *theWeightOfAPerson*, *theBodyMassIndexOfAPerson*, and *theSquareOfANumber*.

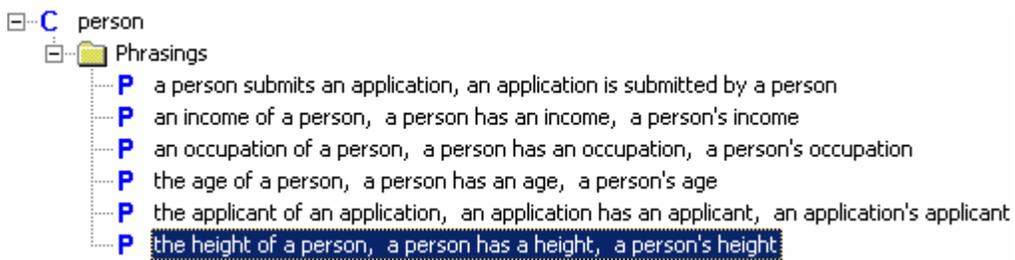
4.8.2.1 *theHeightOfAPerson*, *theWeightOfAPerson*, and *theBodyMassIndexOfAPerson*

Drag the *height* concept on to the *person* concept to open the **Updating the relation's roles and properties** modeling wizard. Type the relation name, *theHeightOfAPerson*, in the **Name** field. Edit the properties of the *person* concept to reflect that the role of *person* in the relation is:

- Always filled
- Is unique
- Is required to participate

The default properties of the concept *height* are correct; there is no need to edit its properties. Click **Next >** to display the **Adding a relation...** screen.

Type the noun phrase *the height of a person* in the **add noun phrasing** field and click **Next >**. The modeling wizard closes and the new phrasing is added to the knowledge tree under the concepts *person* and *height*.

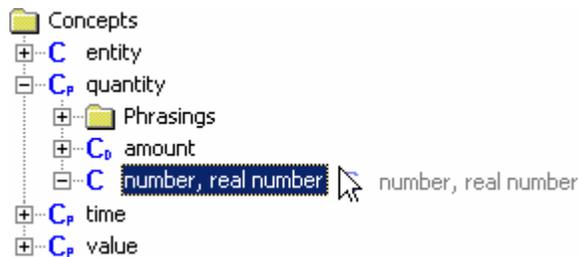




Repeat the same steps to create the relations *theWeightOfAPerson* and *theBodyMassIndexOfAPerson*.

4.8.2.2 *theSquareOfANumber*

The concept used to create the relation *theSquareOfANumber* is *number*. More precisely, it is two numbers – $number * number = number^2$. Therefore, we are going to drag the concept *number, real number* and drop it on itself.

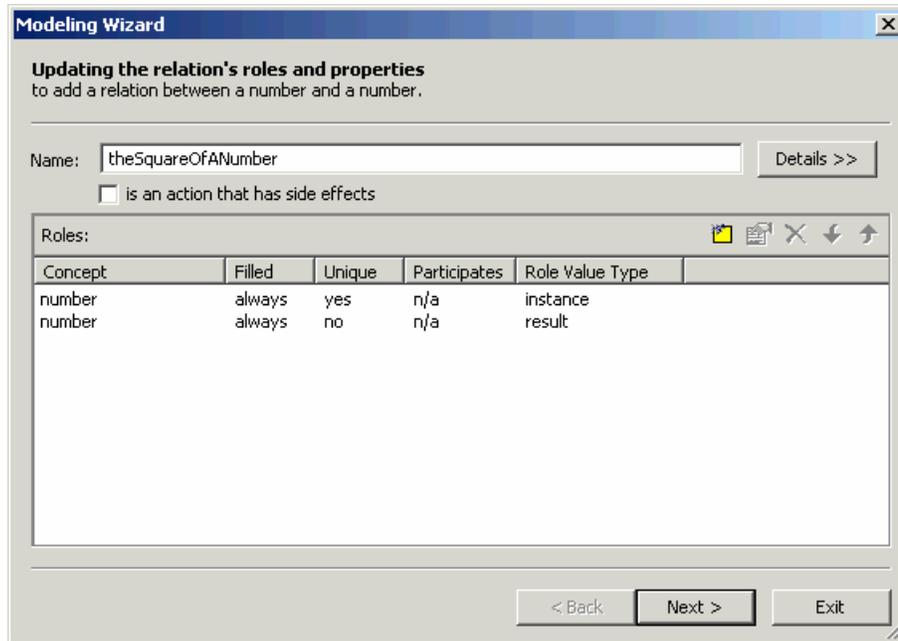


The **Updating the relation's roles and properties** modeling wizard opens. Type the name of the relation, *theSquareOfANumber*, in the **Name** field.

Double-click on the first *number* concept in the **Roles** section of the wizard to open the **Edit role** dialog. This concept will be the number that is multiplied by itself (e.g., $10 * 10$), therefore, we want to check the option *a number participates in at most one such relationship* because a number has only one square. The option every such relationship must have a number is checked by default, leave it checked. Click OK to close the dialog and return to the wizard.

Now we want to change the **Role Value Type** of the other *number* concept so that it is the *result* ($number^2$) of $number * number$. In the **Roles** section of the wizard, double-click on the second *number* concept to open the **Edit role** dialog.

Edit the properties of the concept so that **every such relationship must have a number** and that *number* is **the result of a function or predicate**. Click **OK** to close the dialog and to return to the modeling wizard. The role value type for one of the *number* concepts is a *result*.



Click **Next >**. The **Adding a relation...** screen opens.

Type *the square of a number* in the **add noun phrasing** field and click **Next >**. The modeling wizard closes and the relation is added to the tree.

Let's see how much HaleyAuthority understands about the two statements and conditions now.

a person's body mass index is the person's weight in pounds times 703 divided by the square of the person's height in inches

the square of a number is the number times the number

the body mass index of the application's applicant is not between 20 and 30

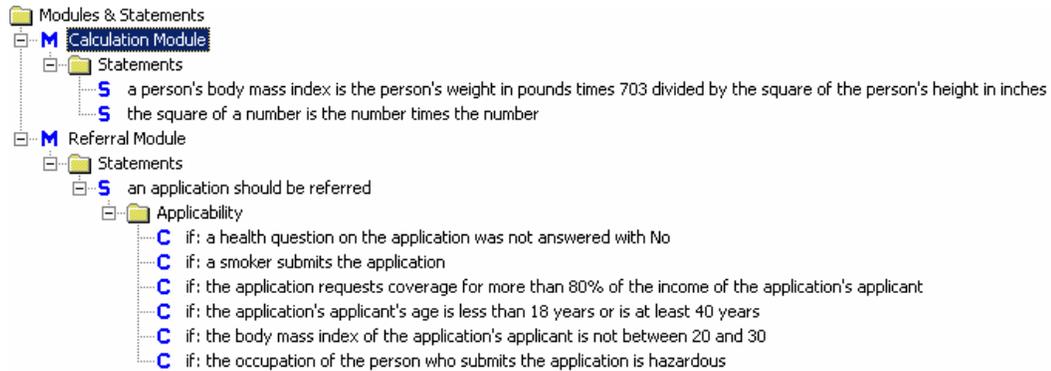
HaleyAuthority seems happy with the three statements. Be sure to visit all three statements and save each one using the **OK** button in the **Edit statement/Edit condition** dialogs.

4.9 Adding the Calculation Module

As a matter of good form, we should create a new module, the **Calculation Module**, as a container for the two statements that calculate the body mass index. These statements create utility rules, not business policies, and they don't belong in the **Referral Module**. Policy makers do not need to see or modify these utility rules.

Right-click on the **Modules & Statements** node of the knowledge tree. Select **Add a module...** Call the new module *Calculation Module*.

Then simply drag the utility rules from the **Referral Module** and drop them in the **Calculation Module**. The result looks like this:



4.10 Final disposition rules

The rules we have defined up to this point either detect policy violations or perform low-level calculations. The two remaining rules are fundamentally different.

- If the application should be referred, then refer the application to the underwriting department.
- If the application should *not* be referred, then approve the application.

These two statements are presumed to have *side effects*; they either approve an application or they refer it to underwriting. When either one of these rules fires, the analysis is ended. The AA Application system records the decision in the database and moves on to the next applicant. (In a real system, these rules would call functions that alter fields in a database or that generate an XML file.)

Rules are data driven, and will try to fire as soon there is appropriate data to fulfill their conditions. We say they try to fire because several rules may be ready at once, but *only one gets to fire*. Each time a rule fires the state of knowledge changes by the insertion or deletion of a fact. This forces the rule engine to reevaluate the list of rules still waiting to fire.

This opportunism is the strength of a rule-based system but it also creates a problem. We can permit the analysis rules to fire opportunistically... in any order at all... as long as they have all the time they need to complete the analysis. These two new rules, however, will prematurely *end* the analysis. They will try to fire as soon as their patterns encounter matching facts. What does this mean in a practical sense?

If the application should be referred, then refer the application to the underwriting department. This rule will try to fire the instant any of the rules determines that an application should be referred. A rule like this one could prevent the firing of other rules prematurely. That wouldn't make much practical difference in the current example, but in general we would like all of the rules to have the opportunity to fire, if appropriate, before taking any permanent action.

What about the other rule? Can you anticipate what it will do? Look closely. *If the application should not be referred*, then approve the application.

Eclipse reacts to the presence or absence of matching facts. The first rule will fire when there exists even a single fact saying that the application should be referred. The other rule will fire when no such fact exists.

The second rule will try to fire the instant *any new application appears*.

The information about a new application does not include a fact recommending referral. Therefore, the approval rule tries to fire immediately. This is clearly a disruptive development.

HaleyAuthority uses statement modules to separate related groups of statements. Each module has a *priority* number, defaulting to zero.

The module's priority setting can be used to force the rules of one module to wait until the rules of another module have finished:

- At any given moment, all of the rules from the priority 0 modules are in equal competition with one another for the opportunity to fire next. The order of firing is opportunistic, and is not under the knowledgebase administrator's control.
- If a rule from a module of priority -100 tries to fire, it will have to wait until all of the higher-priority rules are finished.
- If a rule from a module with priority +100 suddenly matches data, it will instantly go to the front of the line and fire, making all of the lower-priority rules wait their turn.

What is the correct thing to do here? We will create a new module for the two final rules, and give it a negative priority. This forces the last two rules to wait until all of the other rules are finished before firing. That way we won't terminate the analysis prematurely.

How do we select a priority number for a module? The numbers have no inherent significance. The default priority is 0. For a higher-priority module, pick any positive integer. For a lower-priority module, pick any negative integer. It is a good practice to leave some gaps between the numbers you choose, in case you need to create a module of intermediate priority later on.

Right-click on the **Modules & Statements** node and select **Add a module...** Name this module **Final Disposition**. Right-click on the new **Final Disposition** node and select **Properties...** The **Properties of module** dialog opens. Set the module's priority to -100, so that these rules will run only when the other rules have finished. Click **OK** to close the dialog

Add two new statements to this module:

approve an application if the application should not be referred

if an application should be referred then refer the application to the Underwriting Department

The action of the first statement is *imperative*, meaning that it seems to be giving someone an order. An imperative phrasing is a phrasing that has *no explicit subject*. The implied subject is *you*: *(You) approve the application*.

To change a default phrasing into an imperative phrasing, one usually just reassigns the subject of the phrasing to be the direct object.

As an experienced HaleyAuthority user, you recognize immediately that we need a couple of new concepts and relations to support these statements:

- Create the relation **approveAnApplication**. You will need a new verb, *approve*, and you'll need to put **application** in the role of *direct object*. HaleyAuthority knows that a phrasing without an explicit subject is imperative. You should be able to create this on your own. The only difference is that this relation will have side effects; be sure to indicate that the relation will have side effects when you are building the relation in the modeling wizard. Something will happen *outside* of Eclipse when an application is approved.

Edit verb phrasing

Modal:
 Adverb:
 Auxiliary:
 Verb: approve
 Adjective:

Phrasings: approve an application

Roles in phrasing:

Grammatical Role	Preposition	Noun Phrase	Relational Role	Mandatory
direct object		an application	application	yes

OK Cancel

- Add a department entity with an underwriting department instance. Add the relation *referAnApplicationToADepartment*. This relation uses the verb *refer*, with *an application* in the role of *direct object*, and *a department following an application* with the preposition *to*.

Edit verb phrasing

Modal:
 Adverb:
 Auxiliary:
 Verb: refer
 Adjective:

Phrasings: refer an application to a department

Roles in phrasing:

Grammatical Role	Preposition	Noun Phrase	Relational Role	Mandatory
direct object		an application	application	yes
follows an application	to	a department	department	yes

OK Cancel

When the concepts and relations are in place, check the status of the draft statements. If all has gone well, you will be able to save both statements as active rules.

4.11 Scenario

Mike Marks sat back from his keyboard and rubbed his eyes. The process of educating HaleyAuthority for the first time had taken an afternoon of trial and error. Implementing

the first statement had taken significant effort because of the unfamiliar tasks and the extensive infrastructure of entities, values and relations that had to be put in place. Mike had exchanged several email messages with Haley customer support along the way.

The second statement, he found, was not quite as difficult to implement. The third statement took only seconds to input, once the Haley representative told Mike about the cascading menus. Mike found that the effort required to create a new statement fell off very rapidly as HaleyAuthority learned more and more about the insurance domain.

By the time Mike reached the sixth statement, creating a new rule had become routine. When he didn't have an appropriate module, he created one. He typed in the statements and then followed the bold print from left to right, methodically defining any phrase that was unclear to HaleyAuthority. Creating a new entity was trivial. Adding a new attribute to the entity took only seconds.

Mike surveyed his list of statements with satisfaction. He had the full system in place. As promised, every statement had been entered in English. A policy maker could see at a glance how the AA Application system behaved.

"Ok so far," Mike mused to himself. "Let's see if it runs." Somewhere he'd seen a node labeled "test cases." Or maybe it was "regression tests." He reached for the mouse...

5 Applying policy to test cases

HaleyAuthority lets the knowledgebase administrator create test cases and run them, collecting trace messages as the rules activate and fire. You don't have to set up the data communication infrastructure just to see the rules make their first decisions.

First we will need a couple of typical applicants, John and Jane, with characteristics chosen to produce referral (John) and approval (Jane).

Then we'll need to create a test case for the applicants, and finally we'll execute the test case and view the trace messages. We'll verify that the rules make the expected decisions, and we'll study how they interact with one another as they run.

5.1 Scenario

It was almost quitting time when Mike Marks looked up and found the company CEO at the door of his office. "How's did that natural language thing turn out?" asked the CEO.

"I was just about to test it," replied Mike. "Pull up a chair."

Mike gave the executive a quick tour of the AA Application system's entities and statements, and showed him how quickly he could create a new statement just by typing it in.

"Yeah but does it run?" asked the boss. "Otherwise I'll just stick with Notepad."

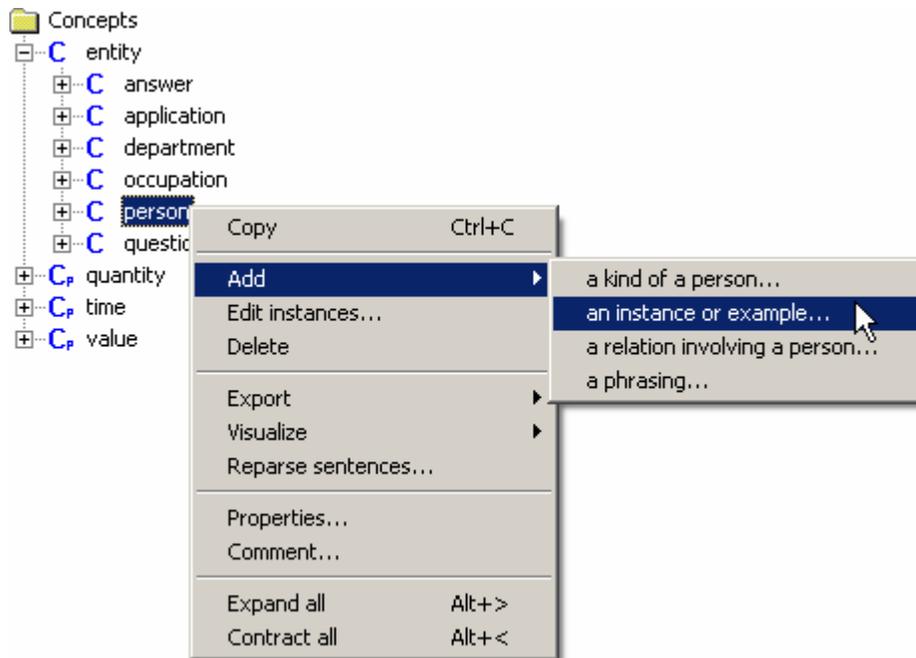
"Let's find out," said Mike. "This is my test case..."

5.2 Applicant instances

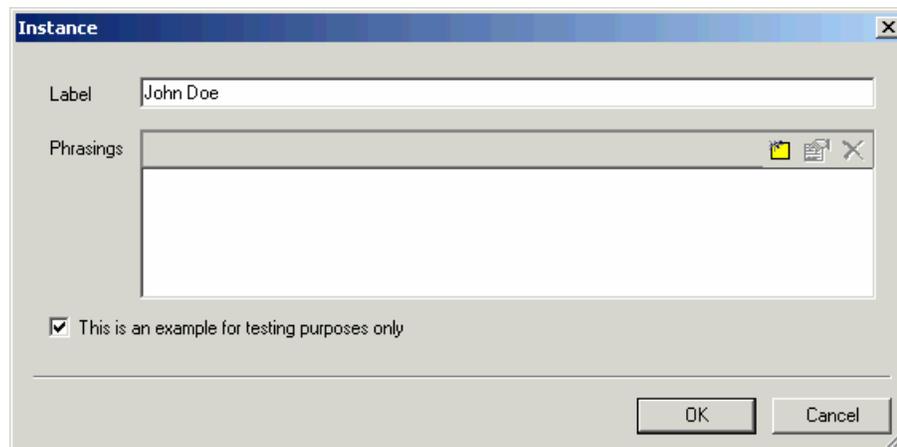
In the final AA Application production system, data will enter Eclipse from records in an SQL database or from an XML file. If that subsystem isn't ready yet, we can manually create some data objects in HaleyAuthority and use them to try out the rules.

5.2.1 John Doe

We begin by creating *John Doe*, who is an instance of a *person*. Right-click on **person** and select **Add > an instance or example...**



The **Instance** dialog opens. Type *John Doe* in the **Name** field of the instance, and be sure to put a checkmark in the **This is an example for testing purposes only** box. Otherwise, the deployed system will diagnose *John Doe* every time you start it up.

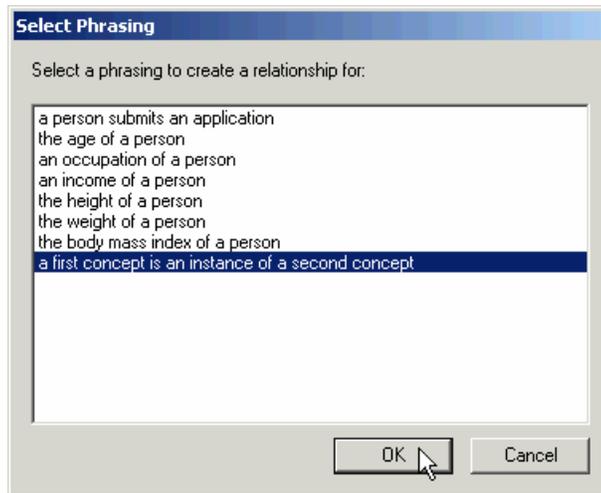


Click **OK** to close the dialog. *John Doe* is now an instance of a *person*.

Right-click on the **John Doe** instance and select **Properties...** from the menu. The **Properties of instance** dialog opens with the **Instance** tab displayed. Click on the **Facts** tab to bring it to the front of the dialog. Now we are going to define the attributes of Mr.

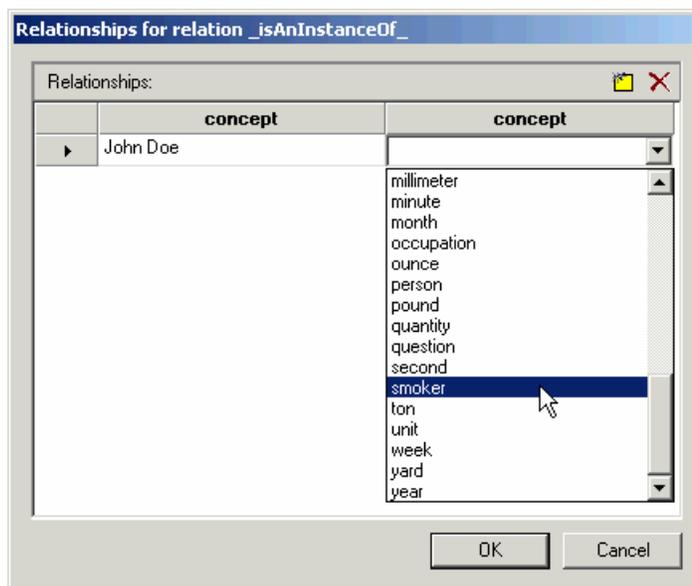
Doe. In the **Relationships** section of the **Facts** tab, click  to open the **Select Phrasings** dialog. This dialog lists the attributes you may assign to John Doe.

While there is no specific phrasing that mentions *smoker*, there is a phrase that, when the appropriate concepts are inserted, will produce the phrasing *John Doe is a smoker*. That phrasing is *a first concept is an instance of a second concept*.



Select that phrasing and then click **OK**. The **Select Phrasings** dialog closes, however, the **Relationships for relation _isAnInstanceOf_** dialog opens in its place.

HaleyAuthority is asking you to fill in the concepts that will build the phrasing. Because we are in the midst of adding facts about *John Doe*, HaleyAuthority has already selected *John Doe* as one of the concepts. Select *smoker* as the second concept.

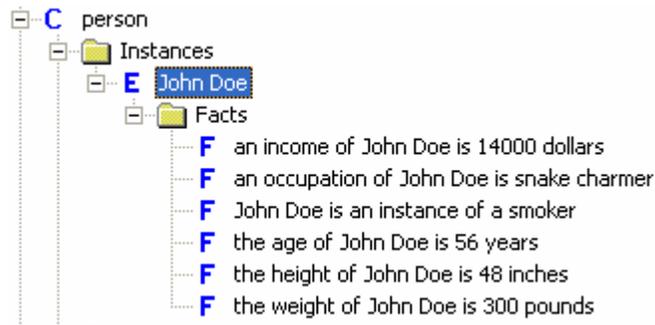


Click **OK** to close the dialog and return to the **Facts** tab. The new fact, *John Doe is a smoker*, is listed on the tab.

In the **Relationships** section of the **Facts** tab, click  again to open the **Select Phrasings** dialog. This time, select *the height of a person* and click **OK**.

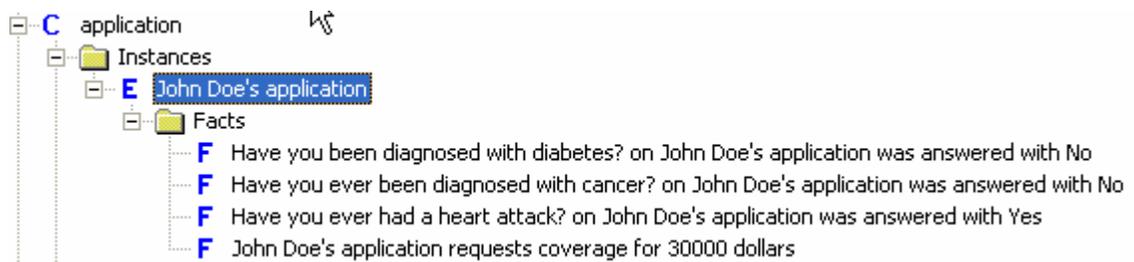
This time HaleyAuthority wants to know how tall John is. John is 48 inches tall. Type in the value and click **OK**.

Now HaleyAuthority knows two facts about John. Using the same procedure, you may go ahead and input all of the values shown in the following illustration. As you can see, John is not a good bet for automatic life insurance approval.

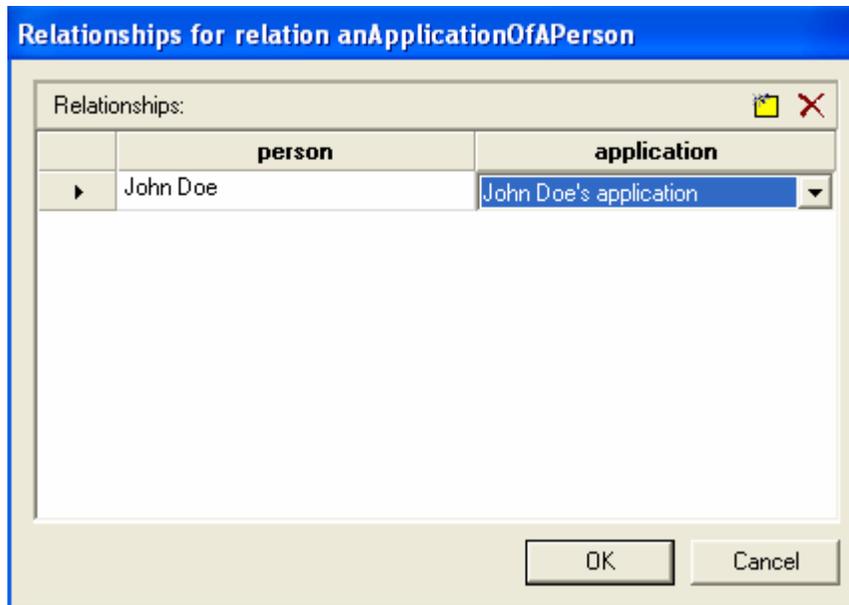


We will also need an instance representing John's application. Close the **Properties of instance** dialog. Right-click on **application** and choose **Add > an instance or example...** from the menu to open the **Instance** dialog. Type *John Doe's application* in the **Name** field and be sure to put a checkmark in the **This is an example for testing purposes only box**. Click **OK** to close the dialog.

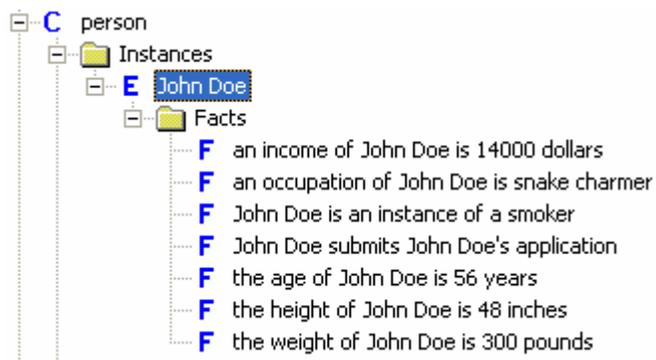
Now we will also need to tell HaleyAuthority some facts about the information included in John Doe's application. Add the following facts to John Doe's application:



Then let's associate John Doe with his application. Right-click on the **John Doe** instance again and select **Properties...** from the menu. The **Properties of instance** dialog opens with the **Instance** tab displayed. Click on the **Facts** tab to bring it to the front of the dialog. Now we are going to add the following facts about John Doe's application.



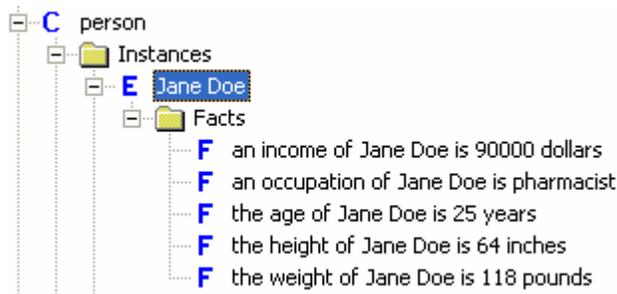
Here are the final facts for John Doe. The **F** icons indicate that these are *facts*. These facts will become Eclipse facts for the rules to match.



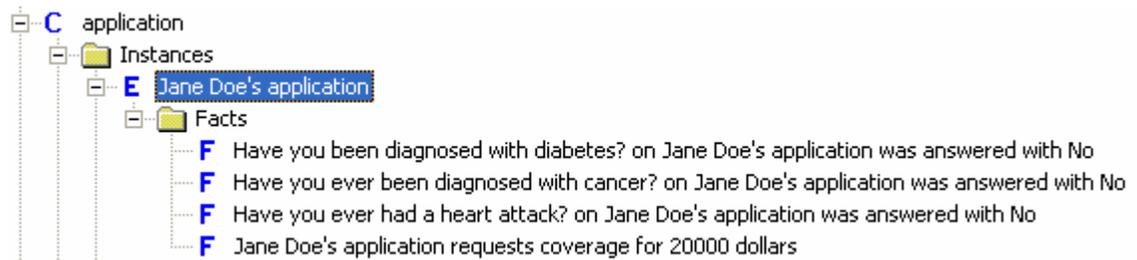
Poor John is too short, too heavy, too old, too poor, and too greedy for automatic approval. *And* he smokes while he handles poisonous reptiles. The AA Application system should send his application to an underwriter like it is on fire.

5.2.2 Jane Doe

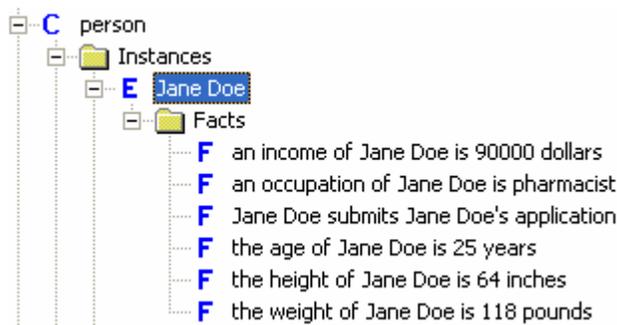
In contrast to John, Jane Doe eats sprouts and plays handball. Here she is. Add Jane Doe as an instance along with her facts:



Don't forget to add Jane Doe's application and the facts for the application. This is Jane's application instance:



Here are the final facts for Jane Doe. Don't forget to add a fact that she submits an application!



The AA Application system will approve Jane in a heartbeat. At least we hope so. She might be too petite. The body mass index has a lower limit for a reason.

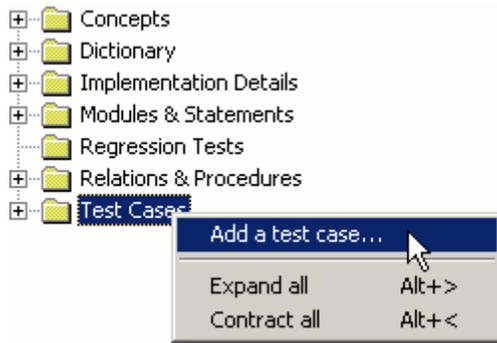
5.3 Running a test case

A test case runs sample data against the current set of rules. It also stores the results of past tests for comparison.

5.3.1 Defining the test case

Look for the **Test Cases** node at the very bottom of the knowledge tree in the **Full View** pane. If you do not see it there, navigate to **View** menu > **Options** > **Full View tab** and check the appropriate box to make this node appear in the tree. Click **OK**.

To add a test case, right-click the **Test Cases** node and select **Add a test case...**



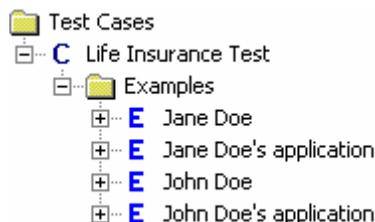
This opens the **Edit test case** dialog. Type *Life Insurance Test* in the **Description** field. In the **Case Data** section of the dialog, the default option is *Examples*, meaning that this case will use example data already defined in HaleyAuthority (the applicant instances). Accept the default **Case Data** setting and click **OK** to close the dialog. **Life Insurance Test** is added as a test case.



The **C** indicates that it is a test **case**.

The next step is to populate the new test case with example data. To do this, drag the **John Doe** instance and drop it on **Life Insurance Test**. The **Add** dialog opens and asks: “Do you want to add the dependent example ‘John Doe’?” Click **Yes**. The **Add** dialog opens again and asks: “Do you want to add the dependent example ‘John Doe’s application’?” Click **Yes** again. You have just added both John Doe and the Facts associated with him and John Doe’s application and the Fact associated with it.

Repeat the same process to add **Jane Doe** and **Jane Doe’s application** to the **Life Insurance Test**.



The **E** indicates example data. Examples are objects that exist only within HaleyAuthority. They are different from Instances, as Instances are generated in the deployment code and can be loaded into the rules engine.

5.3.2 Trace messages

We are going to need some trace messages in order to see what the rules actually do. Pull down the **Tools** menu and select **Options...**

In the **Options** dialog, click on the **Tests & Cases** tab to bring it to the front of the dialog. Select all of the **Watch** items – this means that the trace messages will include all of these types of information. Click **OK** to close the **Options** dialog.

Next, pull down the **View** menu and select **Output**. This opens a special window for viewing the trace messages.

The new window will probably bury the main HaleyAuthority window. You can minimize the **Output** window for now.

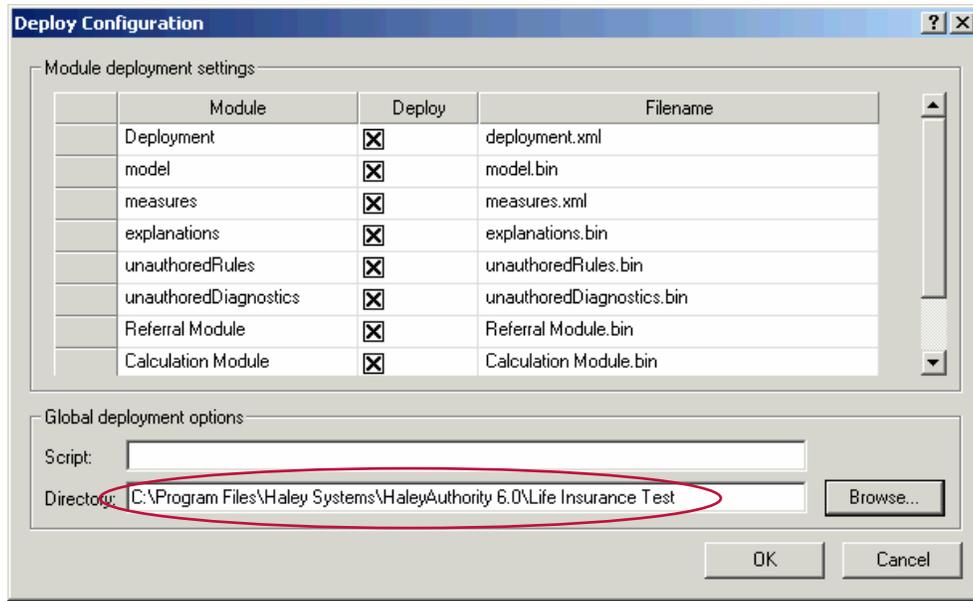
Right-click the **Life Insurance Test** node and select **Test Case** from the menu.

We are about to run the rules for the first time. Before we do, there is some housekeeping to do.

5.3.3 Deploying the logic for a test case

When you select **Test Case**, HaleyAuthority asks if you want to deploy your logic. In other words, have you changed the rules or sample data since the last time you ran this test case? You probably did, or there would be no reason to run the case again, so click **Yes**. The **Deploy Configuration** dialog opens. HaleyAuthority wants to know where to save the rule files.

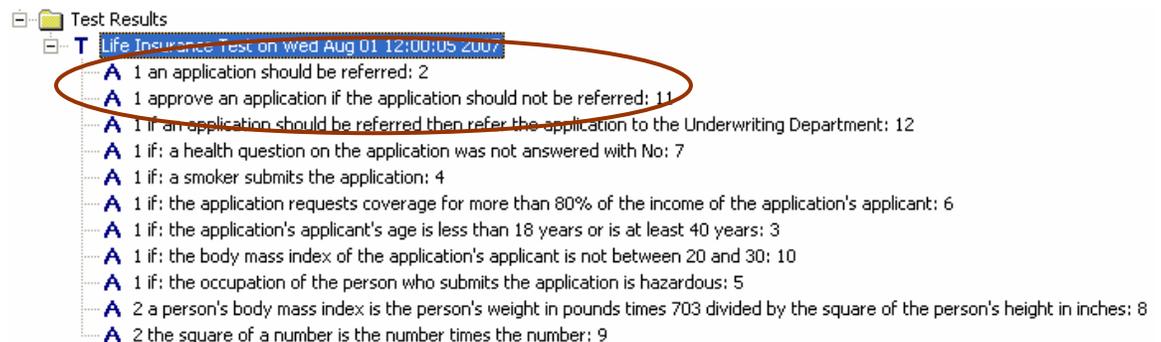
Click **Browse** in the lower right of the dialog and select the directory where you want to save the rule files.



Click **OK**. The Deployment Progress dialog opens, which displays the progress of the test.

5.3.4 Results of the test case

After running the test case, you will find a time-stamped set of results below the **Life Insurance Test** node.



This is a list of the rules (statements) that fired during the test. The **A** icon indicates that these are *applications* of the rules. The number at the beginning of the line is the number of times the rule fired during the test. This display gives you a general indication that a test has executed correctly. For instance, this application shows that one applicant's application (John Doe's application) was rejected for automatic approval, while another applicant's application (Jane Doe's application) was approved.

For more detail, we turn to the **Output** window. Pull down the **Window** menu and select **Output**. We find a solid block of trace messages in the **Output** window, detailing exactly what the rules have done.

Test Case: Life Insurance Test
=> Jane Doe is an instance of an applicant
=> John Doe is an instance of an applicant
=> the weight of Jane Doe is 118 pounds
=> the weight of John Doe is 300 pounds
=> Jane Doe's application requests coverage for 20000 dollars
=> John Doe's application requests coverage for 30000 dollars
=> an income of Jane Doe is 90000 dollars
=> an income of John Doe is 14000 dollars
=> the age of Jane Doe is 25 years
=> the age of John Doe is 56 years
=> the height of Jane Doe is 64 inches
=> the height of John Doe is 48 inches
=> Jane Doe's application is an instance of an application
=> John Doe's application is an instance of an application
=> Jane Doe is an instance of a person
=> John Doe is an instance of a person
=> John Doe is an instance of a smoker
=> firefighter is hazardous
=> an occupation of Jane Doe is pharmacist
=> scuba diver is hazardous
=> an occupation of John Doe is snake charmer
=> snake charmer is hazardous
=> Have you been diagnosed with diabetes? on John Doe's application was answered with No
=> Have you been diagnosed with diabetes? on Jane Doe's application was answered with No
=> Have you ever been diagnosed with cancer? on John Doe's application was answered with No
=> Have you ever been diagnosed with cancer? on Jane Doe's application was answered with No
=> Have you ever had a heart attack? on John Doe's application was answered with Yes
=> Have you ever had a heart attack? on Jane Doe's application was answered with No
=> John Doe submits John Doe's application
=> Jane Doe submits Jane Doe's application
Execute: condition 3: the application's applicant's age is less than 18 years or is at least 40 years .
Execute: condition 4: a smoker submits the application .
Execute: condition 5: the occupation of the person who submits the application is hazardous .
Execute: condition 6: the application requests coverage for more than 80% of the income of the application's applicant .
Execute: condition 7: a health question on the application was not answered with No .
Execute: statement 2: an application should be referred .
=> John Doe's application should be referred
Execute: statement 9: the square of a number is the number times the number .
=> the square of 64 is 4096
Execute: statement 8: a person's body mass index is the person's weight in pounds times 703 divided by the square of the person's height in inches .
=> the body mass index of Jane Doe is 20.252441
Execute: statement 9: the square of a number is the number times the number .
=> the square of 48 is 2304
Execute: statement 8: a person's body mass index is the person's weight in pounds times 703 divided by the square of the person's height in inches .
=> the body mass index of John Doe is 91.536458
Execute: condition 10: the body mass index of the application's applicant is not between 20 and 30 .
Execute: statement 11: approve an application if the application should not be referred .
=> approve Jane Doe's application
Execute: statement 12: if an application should be referred then refer the application to the Underwriting Department .
=> refer John Doe's application to Underwriting department

That's a thick block of small print. Let's take this one step at a time, to see how the rules really work.

=> an income of Jane Doe is 90000 dollars
=> an income of John Doe is 14000 dollars
=> Jane Doe's application requests coverage for 20000 dollars
=> John Doe's application requests coverage for 30000 dollars
=> Jane Doe's application is an instance of an application
=> John Doe's application is an instance of an application
=> Jane Doe is an instance of a person
=> John Doe is an instance of a person

=> John Doe is an instance of a smoker

HaleyAuthority starts to compare the facts about the applicants and applications against the statement and conditions in the Referral Module. At this point, HaleyAuthority has determined that John and Jane's applications are instances of applications (so every fact associated with an application is also associated with Jane and John's application), both applicant's have an income, and both applications include a requested amount of coverage. In addition, Jane and John are both instances of a *person* (so every attribute of a *person* will be an attribute of Jane and John). Finally, John is an instance of a smoker. We see no mention of Jane being a smoker, therefore, John matched the fact, but Jane did not.

Let's continue reviewing the output:

=> firefighter is hazardous
=> scuba diver is hazardous
=> snake charmer is hazardous
=> Have you ever had a heart attack? on John Doe's application was answered with yes
=> Have you ever had a heart attack? on Jane Doe's application was answered with no
=> Have you ever been diagnosed with cancer? on John Doe's application was answered with no
=> Have you ever been diagnosed with cancer? on Jane Doe's application was answered with no
=> Have you been diagnosed with diabetes? on John Doe's application was answered with no
=> Have you been diagnosed with diabetes? on Jane Doe's application was answered with no
=> John Doe submits John Doe's application
=> Jane Doe submits Jane Doe's application

Now HaleyAuthority has figured out which of the six occupations are hazardous (firefighter, scuba diver, and snake charmer) and evaluates John and Jane's answers to the health questions. Of the six questions asked of the two applicants, only one question yielded a *yes* answer – John answered *yes* to the question *Have you ever had a heart attack?* Lastly, HaleyAuthority ascertains the relationship between the applicants and their applications.

Execute: condition 4: a smoker submits the application.
Execute: condition 5: the occupation of the person who submits the application is hazardous.
Execute: condition 7: a health question on the application was not answered with No.
Execute: statement 2: an application should be referred.
=> John Doe's application should be referred

HaleyAuthority now starts executing **Referral Module** rules based upon the facts it has evaluated to this point:

- Because John is an instance of a smoker, the condition *a smoker submits the application* fires.
- Because John's occupation is snake charmer and snake charmer has been identified as a hazardous occupation, thus, John has a hazardous occupation, the condition *the occupation of the person who submits the application is hazardous* fires.

- Because John answered *yes* (not *no*) to one of the health questions, the condition a *health question on the application was not answered with No* fires.

Based upon the firing of the three conditions, the statement that the conditions are attached to (*an application should be referred*) fires. As a result, John Doe's application should be referred. Note that none of the statement's conditions fired when evaluating Jane's application, thus, her application has not yet been recommended for referral at this point.

One of the calculation rules has fired. The square of John's height is calculated to be 2304. Up to this point there has been no chaining. All of these rules were ready to fire from the moment the rule engine began to run. They might have fired in any order. This particular order has no significance.

The *square* calculation, however, starts a chain reaction. Now that the square of John's height is known, his body mass index can be calculated.

John's body mass index is determined to be 91.5365. Now that the body mass index is known, the rules can determine whether it is out of range.

John's body mass index is extremely high, so he should be referred to underwriting.

At this point, the calculation and reference rules have exhausted their matching facts. They are all finished, and the net effect is to trigger the final disposition rule. This rule has a lower priority than the others, and has been waiting patiently ever since the first rule complained that John was a smoker.

John Doe's application has been referred to underwriting.

5.4 Scenario

"Looks like it works," Mike concluded with relief.

His CEO sat back and frowned. "OK, but what if I want to change something? Suppose I decide to refer anyone with a body mass index below 22, for example?"

Mike smiled. "Time me," he said.

*While his boss looked at his Rolex, Mike turned to HaleyAuthority and edited the statement about the applicant's body mass index. He changed 20 to 22 and clicked **OK**. Then he ran the test case again, redeployed the logic, and expanded the results. "Time!"*

The CEO glanced at him in surprise. "Twenty seconds? You revised a running policy in twenty seconds?"

“And tested it, too,” Mike smiled, pointing at the screen. “And this time Jane didn’t get automatic approval. Her application has been routed to underwriting because she is too thin.”

The CEO leaned in to read the trace messages on Mike’s screen. “Sure did,” he admitted. Then he smiled and clapped Mike on the shoulder. “Good work, Mike. This is what I’ve been looking for.”

Mike felt very pleased for a fleeting five seconds. The CEO snapped open his briefcase and extracted a sheaf of notes. “Here’s the next 500 policies from John Stevens,” he said, handing the bundle to Mike. “I’ll need you to demo the prototype to the board at 10 a.m. Monday. See you there.” He got up and walked out. “Wear a tie!”

Mike stared after the departing figure. Then he weighed the stack of paper in his hand.

Five hundred policies. Estimating ten policies per hour... he looked at his watch. Fifty hours would be 7:00 pm Sunday night, if he worked straight through and didn’t sleep. “No problem,” Mike sighed.

5.5 HaleyAuthority and your business

The advantages of using HaleyAuthority are dramatic, and pose obvious benefits to any business:

- If you can write a statement like the ones in this manual, you can program a rule-based policy system. There is no need to master arcane syntax.
- If you can read a statement like the ones we have shown you, you can understand what the policy system really does. Programmers don’t have to trace the code to find out.
- If you can edit a statement like the ones presented here, you can change what the system does. You can implement a critical policy change in a matter of seconds.

HaleyAuthority gives you control over your business policies. It understands and obeys.