Take Wing: Building Ontologies with Tawny-OWL

Phillip Lord

April 27, 2015

Contents

Chapter 1

Introduction

This book introduces ontology building using the OWL2 ontology language, and the Tawny-OWL library. Ontologies are a method for representing knowledge, generally, but not necessarily, about the world around us. It is then possible to check that the representation is consistent, as well as drawing conclusions about new knowledge. They are generally used in complex, knowledge-rich areas of knowledge, including biomedicine.

Many ontology development tools provide a Graphical User Interface, through which the ontology developer adds the various entities involved in building an ontology. However, many ontologies contain large and repetitive sections; for these, ontology development teams often fall back to generating parts of their ontology programmatically. Tawny-OWL takes a different approach where ontology development in a domain-specific language (DSL) embedded in a full programming language. For structurally simple parts of an ontology, the various components of an ontology can be specified using the default convienient and simple Tawny-OWL syntax; for structurally complex parts, new syntax and new patterns can be built, extending the environment as a core part of ontology development.

This form of programmatic ontology development is still young. At the moment, we have used it to produce large ontologies that would have been difficult using any other technique. However, we also hope that we can also support easier integration of knowledge-rich structures into applications, so that ontological data structures can be come a standard part of the programmers toolkit.

1.1 Status

This manual is a work in progress and there are quite a few bits to write yet. Once, it is somewhat more advanced, we will mark up the individual sections with status markers! This file is also available in HTML

1.2 What is an Ontology

Ontologies are about definitions. It is, perhaps, unsurprising therefore that amount ontologists there are quite a few debates about what exactly an ontology is and is not; it is not our intention here to either cover these arguments, nor to give a comprehensive overview of all the uses of the word.

What is generally agreed is that ontologies describe a set of entities, in terms of the relationships between these entities, using any of a number of different relationships. So, for example, we can describe entities in terms of their class relationships – what is true of a superclass is also true of all subclasses. Or we can describe the *partonomic* relationships: the finger is part of the hand, which is part of the foot.

An ontology is also very similar to a taxonomy; however, ontologies place much greater emphasis on their computational properties. This makes ontologies much more suitable for driving applications and code, although this often comes at the cost of human understandability of the ontology. In this document, all the ontologies we talk about are represented using specific language, called OWL (the Ontology Web Language). This has very well-defined computational properties, and through the document we will explore the implications of these properties.

We also use the term "ontology" to mean a specific object that you can manipulate in Tawny-OWL, which is a slightly more constrained use. This is quite common in many books on programming: we hope that the context should be clear.

1.3 Who this book is for

We have two primary audiences for this book. The first is for the ontologist who is interested in Tawny-OWL as the hub of a new environment for ontology development. The second is for the programmer who is interested in using the rich computational representation of a domain that ontologies provide. There is a risk to having two audiences: that we satisfy neither. To avoid this, the book is built from a series of chapters, each of which covers a

discrete topic, either more programmatic or more ontological. It should be possible to read the chapters independently of each other.

This book does not, however, stand alone. While we try to introduce the background material, we do not intend that, for example, this book will serve as an introduction to programming either in general, or specifically in Clojure. There are many good resources available for this. With ontologies, we give more of a background introduction, but again, we assume that you will be willing to read other material to clarify ontology development. Our hope is that we introduce the material well enough that you feel it is worth the time to investigate other resources, and we include pointers where it seems valuable.

Chapter 2

A Rapid Walk-Through

2.1 A Taster

We take a rapid walk-through an ontology to demonstrate the capabilities of Tawny-OWL. As with all the examples in this book, the code in this chapter is complete, therefore, we need to start with a preamble, defining a namespace and performing some imports.

```
(ns take.wing.walk-through
  (:refer-clojure :only [])
  (:use [tawny.owl]
        [tawny.english]
        [tawny.reasoner]))
```

As we discussed in Section **??**, the word ontology has quite a few different meaning, but here we use it to mean a specific computational object; so, before, we do anything else, we start a new empty ontology, which we call `walk_through`; as it happens, we do not need to refer to this object again because it is now set as the default for the rest of this chapter. We also take the opportunity to set our choice of reasoner, in this case HermiT. We will see later how we use this.

```
(defontology walk_through
  :iri "http://purl.org/ontolink/walk_through")

(reasoner-factory :hermit)
```

Ontologies are all about classes, so we now define two classes one called `Book` and one called `TakeWing` which is a subclass of `Book` [1]. Anything that

[1] One of the joys of ontology development is that the ontology development community is rich with arguments about the correct way to model things. Even, with relatively simple

is true of `Book` must also be true of `TakeWing`.

```
(defclass Book)

(defclass TakeWing
  :super Book)
```

Of course, this does not tell us much about `TakeWing` as a book. There are many properties of books, but one of the most informative is the subject of the book. So, we define a new class of `Subject` and introduce a property `about` which we use to relate books and subjects [2].

```
(defclass Subject)

(defoproperty about)
```

Now, we need some subject listings. Of course, there are many of these in existence already, and Tawny-OWL is fully capable of reusing one of these; however, for this simple example, it is not necessary, so we define a small classification of our own. We describe `Bird` and `Ontology` as subclasses of `Subject` and say that they are different (`:disjoint`) and do not overlap. We also describe `TawnyOWL` as part of the `Ontology` subject.

```
(as-subclasses
 Subject
 :disjoint
 (defclass Bird)
 (defclass Ontology))

(defclass TawnyOWL
  :super Ontology)
```

We can now make some basic queries against the statements that we have made to make sure that they all make sense. So, for example, the `subclasses` function lists all of the subclasses of `Book`, or we can use the predicate function `subclass?`. On its own this functionality is enough to build a simple hierarchy.

```
(subclasses Book)
```

---

models it is easy to hit these arguments and, in fact, we have done so here already. There is a strong argument to say that TakeWing is actually an instance of Book rather than a subclass, because there is only one of them. Or, that TakeWing is a class because there are many copies of TakeWing. Or, that it's a metaclass, because sometimes it operates like a class and sometimes an individual. In this book, we try to touch on these arguments, but not get weighed down by them

[2] Strictly, an *object property*, hence the "o". We describe these more fully later

```
;;=> #{#<OWLClassImpl <http://purl.org/ontolink/walk_through#TakeWing>>}
(subclass? Book TakeWing)
;;=> true
(subclass? Subject TawnyOWL)
;;=> true
```

However, the functionality of OWL allows much richer statements than this. We can extend the existing definition of `TakeWing` and state that it is a book that is about `TawnyOWL` and only about `TawnyOWL`.

```
(class
 TakeWing
 :super
 (some-only about TawnyOWL))
```

Now, we can build some *defined* classes. We describe an `OntologyBook` as a `Book` which is about `Ontology`.

```
(defclass
 OntologyBook
 :equivalent
 (and Book
      (some about Ontology)))
```

There are two critical points about this definition. The first is that we had said nothing at all about the relationship between this class and `TakeWing`. We can confirm this by asking about the `subclasses` of `OntologyBook`, and showing that our ontology knows of no ontology books.

```
(subclasses OntologyBook)
;;=> #{}
```

However, this is not quite true. The second critical part of the definition, the use of `:equivalent`. This allows us to use *reasoning* to infer other subclasses. For this we use the `isubclasses` method instead and find that `TakeWing` can be infered to be an `OntologyBook`.

```
(isubclasses OntologyBook)
;;=> #{#<OWLClassImpl <http://purl.org/ontolink/walk_through#TakeWing>>}
```

We can infer that `TakeWing` is a subclass of `OntologyBook` because we have said that an ontology book is one about ontologies and that this book is about Tawny-OWL which is sub-topic of ontologies. Even in this simple example, we need to put together a number of facts to draw this conclusion.

In this case, though, there is some apparent similarity between the definition of `OntologyBook` and `TakeWing` – both of them are look relatively similar, at least once we substitute `Ontology` for `TawnyOWL` in the definition

of TakeWing. Our computational reasoner, however, does not work in this way, and can draw conclusions even when this similarity does not exist. Consider this example where we describe books which are not about birds.

```
(defclass NonBirdBook
  :equivalent
  (and Book
       (not (some about Bird))))

(subclasses NonBirdBook)
;;=> #{}

(isubclasses NonBirdBook)
;;=> #{#<OWLClassImpl <http://purl.org/ontolink/walk_through#TakeWing>>}
```

Here too, we can classify TakeWing. The chain of logic in this case is that TakeWing is about Ontology, that Ontology is different from Bird, and that, therefore, TakeWing is not about Bird which makes it a NonBirdBook.

This ability to infer new knowledge is the meat and drink of computational ontologies. They allow a rich description of the environment with a tightly defined semantics which makes that environment comptutationally accessible. Here, we have only touched on the expressivity of OWL – there are many constructs that we have not shown yet. We have also used this for only for a small ontology, but as the ontologies grow larger, the value increases.

For existing ontology developers, this will familiar ground. Tawny-OWL, however, brings something new to other mechanisms for developing ontologies; that is a fully programmatic environment. As well as the ability to automate any part of ontology development that we choose, this also brings a rich set of highly-developed tools that programmers have been developing and using for many years to develop software in a repeatable, scalable and highly-collaborative way. It is this which we explore next.

2.2 Environment

Tawny-OWL takes a different approach to other ontology development environments. It is not an application, it is just a programmatic library[3]. This has a key advantage over a more traditional ontology editor; rather

[3]Sort of. In other environments, we have argued that Tawny-OWL is an textual application rather than a programmatic library. In reality, it is a bit of both: it is a library which is designed with development rather than manipulation of ontologies as its primary purpose. For the latter, we would have done things rather differently.

than providing a complete environment, Tawny-OWL just recasts ontology development as a form of software development and borrows its entire environment from software development. This means we can reuse the software engineering environment; our experience is that the richness and maturity of software development tools far outweights any loss of specificity to ontology development.

Our hope is that for structurally simple ontologies, Tawny-OWL should be usable by non-programmers, with a simple and straight-forward syntax. In this section, we introduce the core technology and the basic environment that is needed to make effective use of Tawny-OWL, as well as some optional extras.

2.2.1 The OWL API

Tawny-OWL is built using the `http://owlapi.sourceforge.net/`[`OWL` API]. This library is a comprehensive tool for generating, transforming and using OWL Ontologies. It is widely used, and is the basis for the Protege 4 editor[**?**]. Being based on this library, Tawny-OWL is reliable and standard-compliant (or at least as reliable and standard-compliant as Protege!). It is also easy to integrate directly with other tools written using the OWL API, include Protege.

2.2.2 Clojure

Tawny-OWL is a programmatic library build on top of the Clojure language. Tawny-OWL takes many things from Clojure. These include:

- the basic syntax with parentheses and with `:keywords`
- the ability to effectively add new syntax
- the ability to extend Tawny-OWL with patterns
- integration with other data sources
- the test environment
- the build, dependency and deployment tools

In addition, most of the tools and environment that Tawny-OWL use to enable development were built for Clojure and are used directly with little or no additions. These include:

- IDEs or editors used for writing Clojure
- the leiningen build tool

Tawny-OWL inherits a line-orientated syntax which means that it works well with tools written for any programming language; most notable amoung these are version control systems which enable highly collaborative working on ontologies.

Clojure is treated as a programmatic library – the user never starts or runs Clojure, and there is no `clojure` command. Rather confusingly, this role is fulilled by Leiningen, which is the next item on the list.

2.2.3 Leiningen

`http://www.leiningen.org[Leiningen]` is a tool for working with Clojure projects. Given a directory structure, and some source code leiningen will perform many project tasks including checking, testing, releasing and deploying the project. In addition to these, it has two critical functions that every Tawny-OWL project will use: first, it manages dependencies, which means it will download both Tawny-OWL and Clojure; second, it starts a REPL which is the principle means by which the user will directly or indirectly interact with Tawny-OWL.

2.2.4 REPL

Clojure provides a REPL – Read-Eval-Print-Loop. This is the same things as a shell, or command line. For instance, we can the following into a Clojure REPL, and it will print the return value, or 2 in this case.

```
;; returns 2
(+ 1 1)
```

The most usual way to start a REPL is to use leiningen, which then sets up the appropriate libraries for the local project. For example, `lein repl` in the source code for this document, loads a REPL with Tawny-OWL pre-loaded.

In practice, most people use the REPL indirectly through their IDE.

2.2.5 IDE or Editor

Clojure is supported by a wide variety of editors, which in turn means that they can be used for Tawny-OWL. The choice of an editor is a very personal one (I use Emacs), but in practice any good editor will work.

The editor has two main roles. Firstly, as the name suggests it provides a rich environment for writing Tawny-OWL commands. Secondly, the IDE will start and interact with a REPL for you. This allows you to add or remove new classes and other entities to an ontology interactively. Tawny-OWL has been designed to take advantage of an IDE environment; in most cases, for example, auto-completion will happen for you.

2.2.6 Testing

Tawny-OWL can use any of the testing enviroments that come with Clojure, including `clojure.test` which is the most basic environment provided with Clojure. This integrates well with both leiningen or an IDE both of which will run tests for you and report on test cases.

2.2.7 Version Control and Collaboration

Most ontologies are developed by many people, so some form of collaboration support is needed. In general, with Tawny-OWL we achieve this in the same way that programmers do; rather than providing a collaborative environment where multiple people can edit the environment at the same time, we use version control where different developers use slightly different versions of the ontology, and then merge them together at the end. This works well with Tawny-OWL as it has an attractive, line-orientated syntax. The various version control tools can scale easily to thousands of developers which is well in excess of most ontology projects. For this purpose, we use `git`.

2.2.8 Continuous Integration

An ontology can be *continuously integrated* with both other ontologies that it depends on, and with the software environment which uses it. Unlike other ontology continuous integration systems, Tawny-OWL is just a library – so anything that works with Clojure (or more abstractly a Java Virtual Machine) will also work with Tawny-OWL.

2.3 Recap

In this Chapter, we have built a small basic ontology which non-the-less shows the computational power of OWL ontologies, while surveying the

advantages that Tawny-OWL brings as a development environment for ontologies.

Chapter 3

The Pizza Ontology

3.1 Introduction

In this section, we will create a Pizza ontology; we choose pizzas because they are simple, well-understood and compositional (see here for more).

We have described ontologies more abstractly earlier (see Section 1.2). More concretely, in this book, an ontology is a computational object, which can contain a number of different objects. These objects can be of several different kinds. The most (and least!) important of these are *individuals*. We say that these are the most important because it is these individuals that are described and constrained by the other objects. We say that they are the least important because, in practice, many ontologies do not explicitly describe any individuals at all.

If this seems perverse, consider a menu in a pizza shop. We might seem an item described saying "Margherita....5.50". The menu makes no statements at all about an individual pizza. It is saying that any margherita pizza produced in this resturant is going to (or already has) cost 5.50. From the menu, we have no idea how many margherita pizzas have been produced or have been consumed. But, menu is still useful. The menu is comprehensive, tells you something about all the pizzas that exist (at least in one resturant) and the different types of pizza. This is different to the bill, which describes individuals – the pizzas that have actually been provided, how many pizza and how much they all cost. In ontology terms, the menu describes the **classes**, the bill describes individuals [1]. OWL Ontologies built

[1]The analogy between a pizza menu and an ontology is not perfect. With pizza, people are generally happy with the classes (i.e. the menu) and start arguing once about the individuals (i.e. the bill); with ontologies it tends to be the other way around

with Tawny-OWL *can* describe either or both of these entities but in most cases focus on classes.

3.2 Defining Classes

We start with a namespace form, this includes a `:use` statement for `tawny.owl` and a statement declaring a new ontology. First, consider the syntax of this example, because it is shared by all statements in Tawny-OWL. All expresions in Clojure are delimited by `(` and `)` and Tawny-OWL follows this rule. Next, we have a name for the object we wish to create – in this case an new ontology. This starts with `def` to indicate that we also want to create a new symbol which we can use to refer to this entity later.

Finally, come a set of arguments, introduced with *keywords*. These all end with a `:`. In this case, `:iri` introduces the main IRI for this ontology, which is a global identifier, and finally a string which is the actual value of that argument.

```
(ns take.wing.the-pizza-ontology
  (:use [tawny.owl]
        [tawny.reasoner])
  (:refer-clojure :only []))

(defontology pizza :iri "http://purl.org/ontolink/take-wing/pizza")
```

The semantics of this statement are quite interesting. If we had created a new database, by default, the database would be considered to be empty – that is there would be no individuals in it. With an ontology, the opposite is true. By default, we assume that there could be any number of individuals. As of yet, we just have not said anything about these individuals.

Next, we declare two classes. A class is a set of individuals with shared characteristics. For now, we create two classes, `Pizza` and `PizzaComponent`. As with our `defontology` form, have a `def` form; however, in this case, we do not use any arguments. The semantics of these two statements are that, there is a class called `Pizza` and another called `PizzaComponent` which individuals may be members of. However, we know nothing at all about the relationship between an individual `Pizza` and an individual `PizzaComponent`.

```
(defclass Pizza)
(defclass PizzaComponent)
```

To build an accurate ontology, we may wish to describe this relationship further. We might ask the question, can an individual be both a `Pizza` and a `PizzaComponent` at the same time. The answer to this is no, but currently

our ontology does not state this. In OWL terminology, we wish to say that these two classes are *disjoint*. We can achieve this by adding an `as-disjoint` statement.

```
(as-disjoint Pizza PizzaComponent)
```

This works well, but is a little duplicative. If we add a new class which we wish to also be disjoint, it must be added in two places. Instead, it is possible to do both at once [2]. This has the advantage of grouping the two classes together in the file, as well as semantically, which should make the source more future-proof; should we need new classes, we will automatically make them disjoint as required.

```
(as-disjoint
 (defclass Pizza)
 (defclass PizzaComponent))
```

The semantics of these statements are that our ontology may have any number of individuals, some of which may be `Pizza`, some of which may be `PizzaComponent`, but none of which can be both `Pizza` and `PizzaComponent` at the same time. Before we added the `as-disjoints` statement, we would have assumed that it was possible to be both.

As well as describing that two classes are different, we may also wish to describe that they are closely related, or that they are *subclasses*. Where one class is a subclass of another, we are saying that everything that is true of the superclass is also true of the subclass. Or, in terms of individuals, that every individual of the subclass is also an individual of the superclass.

Next, we add two more classes and include the statement that they have `PizzaComponent` as a superclass. We do this by adding a `:super` argument or *frame* to our `defclass` statement. In Tawny-OWL the frames can all be read in the same way. Read forwards, we can say `PizzaBase` has a superclass `PizzaComponent`, or backwards `PizzaComponent` is a superclass of `PizzaBase`. Earlier, we say the `:iri` frame for `defontology` which is read similarly – `pizza` has the given IRI.

As every individual of, for example, `PizzaBase` is `PizzaComponent`, and no `PizzaComponent` individual can also be a `Pizza` this also implies that no `PizzaBase` is a `Pizza`. In otherwords, the disjointness is inherited [3]

[2] In the source code, generated from this book, we are now defining both classes twice, as we have two defclass statements for each. This will actually work okay, although it is not best practice as it is somewhat dependent on the implementation details of the OWL API.

[3] In this ontology, we use a naming scheme using CamelCase, upper case names for classes and, later, lower case properties. As with many parts of ontology development,

```
(defclass PizzaBase
  :super PizzaComponent)
(defclass PizzaTopping
  :super PizzaComponent)
```

As with the disjoint statement, this is little long winded; we have to name the `PizzaComponent` superclass twice. Tawny-OWL provides a short cut for this, with the `as-subclasses` function.

```
(as-subclasses
 PizzaComponent
 (defclass PizzaBase)
 (defclass PizzaTopping))
```

We are still not complete; we asked the question previously, can you be both a `Pizza` and a `PizzaComponent`, to which the answer is no. We can apply the same question, and get the same answer to a `PizzaBase` and `PizzaTopping`. These two, therefore, should also be disjoint. However, we can make a stronger statement still. The only kind of `PizzaComponent` that there are either a `PizzaBase` or a `PizzaTopping`. We say that the `PizzaComponent` class is *covered* by its two subclasses. We can add both of these statements to the ontology also.

```
(as-subclasses
 PizzaComponent
 :disjoint :cover
 (defclass PizzaBase)
 (defclass PizzaTopping))
```

We now have the basic classes that we need to describe a pizza.

3.3 Properties

Now, we wish to describe more about `Pizza`; in particular, we want to say more about the relationship between `Pizza` and two `PizzaComponent` classes. OWL provides a rich mechanism for describing relationships between individuals and, in turn, how individuals of classes are related to each other. As well as there being many different types of individuals, there are can be many different types of relationships. It is the relationships to other classes or individuals that allow us to describe classes, and it is for this reason that the different types of relationships are called *properties*.

---

opinions differ as to whether this is good. With Tawny-OWL it has the fortuitous advantage that it syntax highlights nicely, because it looks like Java.

A `Pizza` is built from one or more `PizzaComponent` individuals; we first define two properties [4] to relate these two together, which we call `hasComponent` and `isComponentOf`. The semantics of this statement is to say that we now have two properties that we can use between individuals.

```
(defoproperty hasComponent)
(defoproperty isComponentOf)
```

As with classes, there is more that we can say about these properties. In this case, the properties are natual opposites or inverses of each other. The semantics of this statement is that for an individual `i` which `hasComponent` `j`, we can say that `j isComponentOf i` also.

```
(as-inverse
 (defoproperty hasComponent)
 (defoproperty isComponentOf))
```

Again, the semantics here are actually between individuals, rather than classes. This has an important consequence with the inverses. We might make the statement that `Pizza hasComponent PizzaComponent`, but this does not allow us to infer that `PizzaComponent isComponentOf Pizza`. Using an every day analogy, just because all bicycles have wheels, we can not assume that all wheels are parts of a bike; we **can** assume that where a bike has a wheel, that wheel is part of a bike. This form of semantics is quite subtle, and is an example of where statements made in OWL are saying less than most people would assume footnote:[We will see examples of the opposite also – statements which are stronger in OWL than the intuitive interpretation].

We now move on to describe the relationships between `Pizza` and both of `PizzaBase` and `PizzaTopping`. For this, we will introduce three new parts of OWL: subproperties, domain and range constraints and property characteristics, which we define in Tawny-OWL as follows:

```
(defoproperty hasTopping
  :super hasComponent
  :range PizzaTopping
  :domain Pizza)

(defoproperty hasBase
  :super hasComponent
  :characteristic :functional
  :range PizzaBase
  :domain Pizza)
```

[4] Actually, two *object* properties, hence defoproperty. We can also define *data* properties, which we will see later

First, we consider sub-properties, which are fairly analogous to sub-classes. For example, if two individuals `i` and `j` are related so that `i hasTopping j`, then it is also true that `i hasComponent j`.

Domain and range constraints describe the kind of entity that be at either end of the property. So, for example, considering `hasTopping`, we say that the domain is `Pizza`, so only instances of `Pizza` can have a topping, while the range is `PizzaTopping` so only instances of `PizzaTopping` can be a topping.

Finally, we introduce a *characteristic*. OWL has quite a few different characteristics which will introduce over time; in this case *functional* means means that there can be only one of these, so an individual has only a single base.

3.4 Populating the Ontology

We now have enough expressivity to describe quite a lot about pizzas. So, we can now set about creating a larger set of toppings for our pizzas. First, we describe some top level categories of types of topping. As before, we use `as-subclasses` function and state further that all of these classes are disjoint. Here, we have not used the `:cover` option. This is deliberate, because we cannot be sure that these classes describe all of the different toppings we might have; there might be toppings which fall into none of these categories.

```
(as-subclasses
 PizzaTopping
 :disjoint
 (defclass CheeseTopping)
 (defclass FishTopping)
 (defclass FruitTopping)
 (defclass HerbSpiceTopping)
 (defclass MeatTopping)
 (defclass NutTopping)
 (defclass SauceTopping)
 (defclass VegetableTopping))
```

When defining a large number of classes at once, Tawny-OWL also offers a shortcut, which we now use to define a large number of classes at once, which is `declare-classes`. While this can be useful in a few specific circumstances, these are quite limited because it does not allow addition of any other attributes at the same time, and in particular labels which most classes will need. In this case, we can generate a lot of classes in a short space, which is useful in a tutorial document.

Neither `MeatTopping` nor `FruitTopping` are declared as `:disjoint` because we have only put a single example.

```
(as-subclasses
 CheeseTopping
 :disjoint

 (declare-classes
  GoatsCheeseTopping
  GorgonzolaTopping
  MozzarellaTopping
  ParmesanTopping))

(as-subclasses
 VegetableTopping
 :disjoint

 (declare-classes
  PepperTopping
  GarlicTopping
  PetitPoisTopping
  AsparagusTopping
  TomatoTopping
  ChilliPepperTopping))

(as-subclasses
 FruitTopping
 (defclass PineappleTopping))

(as-subclasses
 MeatTopping
 :disjoint
 (defclass HamTopping)
 (defclass PepperoniTopping))
```

3.5 Describing a Pizza

And, now finally, we have the basic concepts that we need to build a pizza. First, we start off with a generic description of a pizza; we have already defined the class above, so we want to extend the definition rather than create a new one. We can achieve this using the `class` function:

```
(owl-class Pizza
           :super
```

```
        (owl-some hasTopping PizzaTopping)
        (owl-some hasBase PizzaBase))
```

This introduces several new features of Tawny-OWL:

- this use of `class` requires that `Pizza` already be defined. In other words, we are extending an existing definition. If `Pizza` is not defined, this form will crash.
- a new function `some`
- we create out first *unnamed* classes from a class expression – in this case `(owl-some hasTopping PizzaTopping)`.

The semantics of the last two of these are a little complex. Like a named class (all of those we have seen so far), an unnamed class defines a set of individuals, but it does so by combining other parts of the ontology. The `owl-some` restriction describes a class of individuals with at least one relationship of a particular type. So `(owl-some hasTopping PizzaTopping)` describes the set of all individuals related by the `hasTopping` relationship to at least one `PizzaTopping`. Or alternatively, each `Pizza` must have a `PizzaTopping`. Or, alternatively again, for each `Pizza` there must exist one `PizzaTopping`; it is for this reason that this form of class is also known as an *existential restriction*.

We combine the two statements to say that a `Pizza` must have at least one base and at least one topping. Actually, we earlier defined `hasBase` with the `:functional` characteristic, so together this says that a `Pizza` must have exactly one base.

Finally, we can build a specific pizza, and we start with one of the simplest pizza, that is the margherita. This has two toppings, mozzarella and tomato. The definition for this is as follows:

```
(defclass MargheritaPizza
  :super
  Pizza
  (owl-some hasTopping MozzarellaTopping)
  (owl-some hasTopping TomatoTopping)
  (only hasTopping (owl-or MozzarellaTopping TomatoTopping)))
```

The first part of this definition is similar to `Pizza`. It says that a `MargheritaPizza` is a `Pizza` with two toppings, mozzarella and tomato. The second part of the definition adds two new features of Tawny-OWL:

- `only` a new function which returns a *universal restriction*

- `owl-or` which returns a *union restriction*

The `owl-or` statement defines the set of individuals that is either `MozzarellaTopping` or `TomatoTopping`. The `only` statement defines the set of individuals whose toppings are either `MozzarellaTopping` or `TomatoTopping`. One important sting in the tail of `only` is that it does **NOT** state that these individuals have any toppings at all. So `(only hasTopping MozzarellaTopping)` would cover a `Pizza` with only `MozzarellaTopping`, but also many other things, including things which are not `Pizza` at all. Logically, this makes sense, but it is counter-intuitive [5].

For completeness, we also define `HawaiianPizza` [6].

```
(defclass HawaiianPizza
  :super
  Pizza
  (owl-some hasTopping MozzarellaTopping)
  (owl-some hasTopping TomatoTopping)
  (owl-some hasTopping HamTopping)
  (owl-some hasTopping PineappleTopping)
  (only hasTopping
        (owl-or MozzarellaTopping TomatoTopping HamTopping PineappleTopping)))
```

We can now check that this works as expected by using the `subclass?` and `subclasses` functions at the REPL.

```
take.wing.the-pizza-ontology> (subclass? Pizza MargheritaPizza)
true
take.wing.the-pizza-ontology> (subclasses Pizza)
#{#<OWLClassImpl <http://purl.org/ontolink/take-wing/pizza#HawaiianPizza>>
  #<OWLClassImpl <http://purl.org/ontolink/take-wing/pizza#MargheritaPizza>>}
```

3.6 A simple pattern

The last definition is rather unsatisfying for two reasons. Firstly, the multiple uses of `(owl-some hasTopping)` and secondly because the toppings are duplicated between the universal and existential restrictions. Two features of Tawny-OWL enable us to work around these problems.

[5] Except to logicians, obviously, to whom it all makes perfect sense.

[6] Pizza names are, sadly, not standardized between countries or resturants, so I've picked on which is quite widely known. Apologies to any Italian readers for this and any other culinary disasters which this book implies really are pizza.

Firstly, the `owl-some` function is *variadic* and take a single property but any number of classes. We use this feature to shorten the definition of `AmericanPizza`.

```
(defclass AmericanPizza
  :super
  Pizza
  (owl-some hasTopping MozzarellaTopping
            TomatoTopping PepperoniTopping)
  (only hasTopping (owl-or MozzarellaTopping TomatoTopping PepperoniToppi
```

The single `owl-some` function call here expands to three existential restrictions, each of which becomes a super class of `AmericanPizza` – mirroring the definition of `HawaiianPizza`.

This definition, however, still leaves the duplication between the two sets of restrictions. This pattern is frequent enough that Tawny-OWL provides special support for it in the form of the `some-only` function, which we use to define the next pizza.

```
(defclass AmericanHotPizza
  :super
  Pizza
  (some-only hasTopping MozzarellaTopping TomatoTopping
             PepperoniTopping ChilliPepperTopping))
```

The `some-only` function is Tawny-OWL's implementation of the *closure* axiom. Similarly, the use of `:cover` described earlier implements the *covering* axiom. These are the only two patterns which are directly supported by the core of Tawny-OWL (i.e. the namespace `tawny.owl`). In later sections, though, we will see how to exploit the programmatic nature of Tawny-OWL to build arbitrary new patterns for yourself.

3.7 Defined Classes

So far all of the classes that we have written are *primitive*. Rather than a statement about complexity, this means that as they stand, they cannot be used to infer new facts. So, for example, we know that a individual `MargheritaPizza` will have a `MozzarellaTopping` and a `TomatoTopping`, but given an arbitrary pizza we cannot determine whether it is a margherita. Or, mozzarella and tomato toppings are *necessary* for a margherita, but they are not sufficient.

Defined classes allow us to take advantage of the power of computational reasoning. Let us try a simple example:

```
(defclass VegetarianPizza
  :equivalent
  (owl-and Pizza
           (only hasTopping
                 (owl-not (owl-or MeatTopping FishTopping)))))
```

Here, we define a `VegetarianPizza` as a `Pizza` with only `MeatTopping` or `FishTopping`. The two key point about this definition is that we have marked it as `:equivalent` rather than `:super` and that there is no stated relationship between `VegetarianPizza` and `MargheritaPizza`. We can confirm this at the shell.

```
(subclasses VegetarianPizza)
=> #{}
(subclass? VegetarianPizza MargheritaPizza)
=> false
```

However, now let us ask the same question of a reasoner. First, we choose a reasoner to use (in this case HermiT), and then ask the same questions of Tawny-OWL but now using the versions of functions prefixed with an `i` (for inferred). Now, we see a different result. A `MargheritaPizza` is a subclass of `VegetarianPizza`.

```
(reasoner-factory :hermit)
=> #<ReasonerFactory org.semanticweb.HermiT.Reasoner$ReasonerFactory@4b8a8782>
(isubclasses VegetarianPizza)
=> #{#<OWLClassImpl <http://purl.org/ontolink/take-wing/pizza#MargheritaPizza>>}
(isubclass? VegetarianPizza MargheritaPizza)
=> true
```

The reasoner can infer this using the following chain of logic:

- `MargheritaPizza` has only `MozzarellaTopping` or `TomatoTopping`
- `MozzarellaTopping` is a `CheeseTopping`
- `TomatoTopping` is a `VegetableTopping`
- `CheeseTopping` is disjoint from `MeatTopping` and `FishTopping`
- Likewise, `TomatoTopping` is not a `MeatTopping` or `FishTopping`
- Therefore, `MargheritaPizza` has only toppings which are not `MeatTopping` or `FishTopping`.

- A `VegetarianPizza` is any `Pizza` which has only toppings which are not `MeatTopping` or `FishTopping`.
- So, a `MargheritaPizza` is a `VegetarianPizza`.

Even for this example, the chain of logic that we need to draw our inference is quite long. The version of the pizza ontology presented here is quite small, so while we can follow and reproduce this inference easily by hand for this ontology, for a larger ontology it would be a lot harder, especially, when we start to make greater use of the expressivity of OWL.

Many of the statements that we have made about pizza's are needed to make this inference. For example, if we had not added `:disjoint:` to the subclasses of `PizzaTopping`, we could not make this inference; even though we would know that, for example, a `MozzarellaTopping` was a `CheeseTopping`, by default, the reasoner would not assume that `CheeseTopping` was not a `MeatTopping`, since these two could overlap. There are also some statements in the ontology that we do not use to make this inference. For example, the reasoner does not need to know that a `MargheritaPizza` actually has a `MozzarellaTopping` (the statement `(some hasTopping MozzarellaTopping)`, just that if the pizza has toppings at all, they are only mozzarella or tomato. The semantics of OWL can be subtle, but allow us to draw extremely powerful conclusions.

3.8 Recap

In this chapter, we have described:

- The basic syntax of Tawny-OWL
- New ontologies are created with `defontology`
- Ontologies consist of classes and properties
- Classes describe a set of individuals
- Properties describe relationships between individuals
- Defined classes allow us to make inferences using comptutational reasoning.

In addition, we have introduced the following semantic statements:

- Subclass relationships

- Disjoint classes
- Covering axioms
- Inverse properties
- Domain and range constraints
- Functional characteristics
- `some` and `only` restrictions, and the `some-only` pattern
- `or` and `not` restrictions

Chapter 4

Highly Patterned Ontologies

Many ontologies contain patterns—that is collections of classes and properties which occur repetitive through the ontology. Dealing with this in many ontology development environments is painful. Tawny-OWL is a fully programmatic environment, however. Patterns are dealt with by writing functions and passing parameters; in otherwords, the same way that we deal with code duplication more generally. In this chapter, we will explore how.

4.1 Dealing with Patterns

Some ontologies have very few patterns; all the classes and objects are unique. These ontologies tend to be very small, however. Most ontologies describe many similar things with just a few details differing between them. In this chapter, we use the amino-acid ontology – this describes the chemical constituants that make up proteins. There are twenty of these and they are all very similar, with the same properties.

Graphical tools can provide a partial solution to this problem, by supporting the building of these patterns. For instance, Protege had "wizards" to build various patterns. In fact, the first version of the amino-acid ontology was built to demonstrate one of these patterns [**?**]. This requires extension of the editor for every new pattern, which is acceptable for some generic patterns which can be widely reused, but works badly for patterns with a narrow scope.

An alternative is to use a language like OPPL [**?**], which can directly specify patterns and transformations to ontologies. However, this requires the use of two syntaxes or environments – one for "normal" ontological

code, and one for patternised. It also presents a maintainence problem – the normal and patternised code is intertwinned, so updating a pattern is difficult.

Tawny-OWL take an alternative approach. Instead of providing an alternative language like OPPL, all ontological statements are written in Clojure, which is, itself, an full programming languages. Patterns can be built straight-forwardly by writing or using functions; this can be done in a general library for generic patterns, shared between ontologies. Or, alternatively, it can be done specifically for individual ontologies, in the same syntax, files and development environment as the normal parts. Updates cease to be a problem; in the worst case scenario, this requires restarting the clojure process. Often it does not require even that. In short, with Tawny-OWL patterns become an integral part of ontology development, rather than an external imposition.

In this chapter, we first describe how to use an pre-existing pattern provided by Tawny-OWL, then how to modify this slightly for the amino-acid ontology. Finally, we show how to create a *de novo* patternised section creating several hundred defined classes.

4.2 Creating the Amino Acid Ontology

First, we start with a namespace declaration. This is slightly different from ones used before, as it also `require`s two new namespaces. `tawny.pattern` provides pattern support and one key pattern which forms the core of the amino-acid ontology; `clojure.string` provides string maninpulation capabilities which we will use. We also define the new ontology.

```
(ns take.wing.amino-acid
  (:require [clojure.string])
  (:use [tawny.owl]
        [tawny.pattern]
        [tawny.reasoner]))

(defontology aao
  :iri "http://www.purl.org/ontolink/aao")
```

First, to explain the domain. Proteins are polymers made up from amino-acid monomers. They consist of a central carbon atom, attached to a carboxyl group (the “acid” amino) and amine group (the “amino” group) a hydrogen and an R group. The R group defines the different amino acids. The different R groups have different phyiscal or chemical properties, such

as their degree of hydrophobicity. We call these different characteristics `RefiningFeatures`.

```
(defclass AminoAcid)

(defclass RefiningFeature)
(defclass PhysicoChemicalProperty :super RefiningFeature)
```

There are a number of different ways of measuring hydrophobicity; in reality, it is a continuous property rather than a discrete one, but these are hard to model ontologically. One simple solution to this problem is the *value partition* – we just pick a set of discrete values into which we partition the range. It is the same trick that is used to describe the colours of the rainbow; we force a continuous range into seven colours. Hydrophobicity splits into just two – hydrophobicic and hydrophilic.

The full representation of this knowledge as a value parition is fairly complex. First, we define a root class and an object property, with an appropriate domain and range, and declared functional, as one object can be hydrophilic or hydrophobicic but not both.

```
(defclass Hydrophobicity :super PhysicoChemicalProperty)

(defoproperty hasHydrophobicity :domain AminoAcid
  :range Hydrophobicity :characteristic :functional)
```

Next we need to define the partition values. We make `Hydrophilic` disjoint from `Hydrophobic`. We cannot do the inverse because Tawny-OWL asks us to define classes before using them [1]

```
(defclass Hydrophobic :super Hydrophobicity)
(defclass Hydrophilic :super Hydrophobicity :disjoint Hydrophobic)
```

Finally, we `refine` the first partition `Hydrophobic` to also be disjoint with `Hydrophilic` and then add a covering axioms to `Hydrophobicity`.

```
(refine Hydrophobic
  :disjoint Hydrophilic)

(refine Hydrophobicity
  :equivalent (object-or Hydrophilic Hydrophobic))
```

The use of disjoints and covering axioms is so common that Tawny-OWL provides specific support for adding these, in a way which also allows us to avoid the necessity for refining classes after creation. This produces a much neater definition and is a simple example of the use of patterns.

[1] Actually, it doesn't and it can be avoided just by using strings. But this opens us to spelling mistake errors, and there is a better way to avoid this problem

```
(as-subclasses
   (defclass Hydrophobicity :super PhysicoChemicalProperty)
   :disjoint :cover
   (defclass Hydrophobic)
   (defclass Hydrophilic))

(defoproperty hasHydrophobicity :domain AminoAcid
   :range Hydrophobicity :characteristic :functional))
```

Tawny-OWL, however, allows us to go further with the use of the `defpartition` macro, which allows to specific all the appropriate values at once. It will produce the same axioms as the statements above.

```
(defpartition Hydrophobicity
  [Hydrophobic Hydrophillic]
  :comment "Part of the Hydrophobicity value partition"
  :super PhysicoChemicalProperty
  :domain AminoAcid)
```

`defpartition` is a generic pattern and is not specific at all to the amino-acid ontology. In general, it will serve well, but for the amino-acid ontology we need to define a series of further value partitions. They all have the same super class and domain. It would be nice to create a *localised* pattern which hard-codes these values. As `defpartition` is a macro this is slightly more complex than a normal function, but not heavily so. This macro is unlikely to be of use in another ontology because of these hard-coded values, but it is valuable because it saves typing here and safe-guards us against future changes. Being in the same environment, it is easy to do, so we might as well!

```
(defmacro defaapartition [& body]
  `(defpartition
     ~@body :super PhysicoChemicalProperty
     :domain AminoAcid))
```

The next value partition is as a result somewhat smaller, as it now longer needs to describe the super class and domain. The size value partition is self-explanatory enough; again, this could be related to a continuous physical measurement (such as size in Daltons), but this is not necessary here.

```
(defaapartition Size
  [Small Tiny Large]
  :comment "The physical size of the amino acid.")
```

Finally, we create three more value partitions describing `Charge`, `SideChainStructure` and `Polarity`.

```
(defaapartition Charge
  [Negative Neutral Positive]
  :comment "The charge of an amino acid.")

(defaapartition SideChainStructure
  [Aliphatic Aromatic]
  :comment "Does the side chain contain rings or not?")

(defaapartition Polarity
  [Polar NonPolar]
  :comment "Whether there is a polarity across the amino acid.")
```

Next, we define a set of annotation properties. Amino-acids have a long name, such as `Alanine`, and two shorter names – a three letter abbreviation such as `Ala` and finally one letter abbreviation which is shorter, but harder to remember, in this case `A`.

```
;; annotation properties
(defaproperty hasLongName)
(defaproperty hasShortName)
(defaproperty hasSingleLetterName)
```

Now, we move onto the heart of this amino-acid ontology which is the function which defines a single amino-acid. This is a fairly large definition, but it is fairly repetitive in itself. First we start with the function definition, combined with a few small pre-conditions; these are probably unnecessary in this case, for reasons we will see soon.

```
(defn amino-acid
  "Define a new amino acid. Names is a vector with the long, three letter and
  single amino acid version. Properties are the five value partitions for each
  aa, as a list."
  [names properties]
  {:pre [(= 3 (count names))
         (= 5 (count properties))]}
```

The main part of the amino acid pattern is defined in the next section. The pattern is not that complex – we simply give an amino-acid five properties and three names. We haven't done much error checking to see whether properties are in the right order; this is because errors would be picked up by reasoning, and are anyway unlikely for reasons that should become obvious later. This is done inside a `let` block because we want to capture the return value. This is not strictly necessary as the return value is used only once, but in this case, I think, it increases readability.

```
  (let [aa (owl-class (first names)
```

```
                    :super AminoAcid
                    ;; we have don't test the values are correct here
                    ;; because the code layout should make the order ob
                    ;; and the range constraints should protect us duri
                    ;; reasoning.
                    (owl-some hasCharge (nth properties 0))
                    (owl-some hasHydrophobicity (nth properties 1))
                    (owl-some hasPolarity (nth properties 2))
                    (owl-some hasSideChainStructure (nth properties 3))
                    (owl-some hasSize (nth properties 4))
                    :label (first names)
                    :annotation
                    (annotation hasLongName (nth names 0))
                    (annotation hasShortName (nth names 1))
                    (annotation hasSingleLetterName (nth names 2)))]
```

The last part is not part of the pattern itself. Rather it adds support for *interning*; this is the process by which OWL objects are bound to Clojure symbols. The practical upshot of this is that we (or anyone importing the amino acid ontology) will be able to refer to amino acids using names like `Alanine` rather than being required to use strings inside quotes — `"Alanine"`. This adds (considerable) complexity to the Tawny-OWL definition of the amino-acid ontology, but is probably worth it for ease of downstream use.

To achieve this, we need to return instances of the `tawny.pattern.Named` class, combined with the strings we use to refer to them. In this case, a single amino-acid class gets three names – this is rather unusual but makes sense here.

```
    ;; and return types for intern
    (map ->Named
         names
         (repeat aa))))
```

We could stop here in terms of generating our ontology. However, here we take two more steps, one mostly to make the input more consistent, so that we would see errors easily, and one to make the amino-acid ontology more usuable within the Tawny-OWL environment.

Firstly, we define a function which takes a number of different amino-acid definitions and runs the amino-acid function over them. It then flattens the list of lists that is returned.

```
(defn amino-acids
  [& definitions]
  (apply
   concat
```

```
  (map
   (fn [[names props]] (amino-acid names props))
   (partition 2 definitions))))
```

Finally, we define a macro. This does two things for us. Firstly it provides the convienience of using "bear" words: so `Alanine` instead of `"Alanine"` within the macro itself. A small convienience for a single amino-acid, but a bigger one for all twenty. The `name-tree` macro simply converts an arbitrary data structure containing symbols to the same structure with strings. And, secondly, we *intern* the `Named` values turned from the `amino-acid` function; that is we create a new variable, identified by relevant symbol, with a value which is an OWL entity. The practical upshort of this is that later, we can refer to, again `Alanine` (or `Ala` or `A`) rather than having to use quotes. In terms of the amino-acid ontology itself, this is unnecessary, but it is useful for another ontology importing the amino-acid ontology, so it is worth doing here. In addition and probably more importantly than the convienience, this also provides a degree of safety: attempts, for instance, to refer to an amino-acid `B` will fail with an error as this amino-acid does not exist.

```
(defmacro defaminoacids
  [& definitions]
  `(tawny.pattern/intern-owl-entities
    (apply amino-acids
           (tawny.util/name-tree ~definitions))))
```

Finally, we define all the amino-acid. These have been laid out in alphabetical order, and the properties arranged in a table which means that we can visually check that everything is correct and nothing is missing.

```
(defaminoacids
  [Alanine       Ala A] [Neutral  Hydrophobic NonPolar Aliphatic Tiny]
  [Arginine      Arg R] [Positive Hydrophilic Polar    Aliphatic Large]
  [Asparagine    Asn N] [Neutral  Hydrophilic Polar    Aliphatic Small]
  [Aspartate     Asp D] [Negative Hydrophilic Polar    Aliphatic Small]
  [Cysteine      Cys C] [Neutral  Hydrophobic Polar    Aliphatic Small]
  [Glutamate     Glu E] [Negative Hydrophilic Polar    Aliphatic Small]
  [Glutamine     Gln Q] [Neutral  Hydrophilic Polar    Aliphatic Large]
  [Glycine       Gly G] [Neutral  Hydrophobic NonPolar Aliphatic Tiny]
  [Histidine     His H] [Positive Hydrophilic Polar    Aromatic
Large]
  [Isoleucine    Ile I] [Neutral  Hydrophobic NonPolar Aliphatic Large]
  [Leucine       Leu L] [Neutral  Hydrophobic NonPolar Aliphatic Large]
  [Lysine        Lys K] [Positive Hydrophilic Polar    Aliphatic Large]
  [Methionine    Met M] [Neutral  Hydrophobic NonPolar Aliphatic Large]
```

```
  [Phenylalanine Phe F] [Neutral  Hydrophobic NonPolar Aromatic
Large]
  [Proline       Pro P] [Neutral  Hydrophobic NonPolar Aliphatic Small]
  [Serine        Ser S] [Neutral  Hydrophilic Polar    Aliphatic Tiny]
  [Threonine     Thr T] [Neutral  Hydrophilic Polar    Aliphatic Tiny]
  [Tryptophan    Trp W] [Neutral  Hydrophobic NonPolar Aromatic
Large]
  [Tyrosine      Try Y] [Neutral  Hydrophobic Polar    Aromatic
Large]
  [Valine        Val V] [Neutral  Hydrophobic NonPolar Aliphatic Small]
  )
```

Finally, we clean up by ensuring that all aminoc-acids are disjoint from each other. We could do this earlier in the `amino-acids` function, but as this function only needs to be run once, it makes little difference.

```
(apply as-disjoint (subclasses AminoAcid))
```

As well as demonstrating the use of the value partition one of the motivations behind the amino-acid ontology is to show how we can reason over defined classes. For example, consider this class definition. If we create this and reason over it, we find ten subclasses – all of the amino-acids we have described to be large above.

```
(defclass LargeAminoAcid
  :equivalent (owl-some hasSize Large))
```

This is fine, of course, but is also very slow, as there are a lot of potential classes that we could create. As well as one for each of the twelve values in our five value partitions, we also need all of the permutations of these, which makes quite a few classes.

Of course, being fully programmatic, calculating permutations in Tawny-OWL is a simple enough task; so, why not build all of these defined classes programmatically?

First, we start with a function which given a partition value returns the relevant property. This is rather an unsatisfying solution, as we created these all together; in a future version of Tawny-OWL the pattern class may provide some way to group the elements together.

```
(defn property-for-partition [partition-value]
  (let [partition
        (first (direct-superclasses partition-value))
        op
        (.getObjectPropertiesInSignature aao)]
    (first
     (filter
```

```
      #(= partition
          (first (.getRanges % aao)))
      op))))
```

Next we need a definition for a defined class; this will take a list of partition values. The pattern simply involves making existential (`owl-some`) restrictions to all of the partition values using the appropriate object property. We form the name of the class from all of the partition values.

```
(defn amino-acid-def [partition-values]
  (let [name
        (str
         (clojure.string/join
          (map
           #(.getFragment
             (.getIRI %))
           partition-values))
         "AminoAcid")
        exist
        (map
         (fn [val]
           (owl-some
            (property-for-partition val)
            val))
         partition-values)]
```

Then finally we create the class and package it with its name. As with our previous amino-acid definition, this function has a return value which would allow it to be used to intern the classes created, although we do not actually use that facility here.

```
    (->Named
     name
     (owl-class
      name
      :label name
      :equivalent
      (owl-and AminoAcid exist)))))
```

Calculating a cartesian product is relatively easy in Clojure using the swiss-army knife `for` list comprehension.

```
(defn cart [colls]
  (if (empty? colls)
    '(())
    (for [x (first colls)
          more (cart (rest colls))]
```

```
      (cons x more))))
```

We combine all of these together to create all of the defined classes.

```
;; build the classes
(doall
 (map
  amino-acid-def
  ;; kill the empty list
  (rest
   (map
    #(filter identity %)
    ;; combination of all of them
    (cart
     ;; list of values for each partitions plus nil
     (map
      #(cons nil (seq (direct-subclasses %)))
      ;; all our partitions
      (seq (direct-subclasses PhysicoChemicalProperty))))))))
```

Finally, we check to see whether everything has worked. For this, we will need to use a reasoner, so first we choose a reasoner and check the consistency of our ontology.

```
(reasoner-factory :hermit)
(consistent?)
```

We can also investigate the classes that we have created. None of the created classes should have any asserted subclasses, which we can check.

```
(subclasses
 (owl-class "SmallAminoAcid"))
```

However, we see a totally different picture with the reasoner. We can first check for inferred subclasses.

```
(isubclasses
 (owl-class "SmallAminoAcid"))
```

We might have expected to just see a few as there are only 20 amino-acids, but actually, there are 113 of them. The reason for this is that the reasoner determines the subclass relationships between the defined classes as well as with the named amino-acids: so, for instance, an `HydrophobicSmallAminoAcid` is necessarily also a `SmallAminoAcid` so appears as a subclass. This demonstrates the power of using a computational reasoner; while the conclusions that it reaches are not, in this case, difficult to calculate by hand, with so many classes they would be laborious.

Unfortunately, in this case, they also hide the answer that we are really interested in. In a less programmatic tool, we would be stuck, but this is not a problem in Tawny-OWL; we just filter the defined classes from the result as follows.

```
(filter
 #(not (.isDefined % aao))
 (isubclasses
  (owl-class "SmallAminoAcid")))
```

And the end result? There are six small amino-acids!

With Tawny-OWL it is straight-forward to implement new patterns building a very large number of classes at once; the amino acid ontology is a nice example of this. At the current time, we do not really know how common the requirement is for this sort of ontology; most ontologies in existance are not heavily patternized. But, then, perhaps this is part because the tools for generating patterns were not integrated into our ontology development process; patternized ontologies are not common because they are just too painful to produce.

Even aside from heavily patternized ontologies, this chapter also shows that Tawny-OWL can be easily extended even within the scope of a single ontology. The `defaapartition` macro is only useful here. But, it is easy to write, reduces duplication and increases consistency of the end ontology. Most ontologies have this form of repetition. With Tawny-OWL, managing this repetition becomes the task of the computer and not the task of the human, which is as it should be.

4.3 Recap

In this chapter, we have described:

- The `tawny.pattern` namespace.
- The Value-Partition design pattern
- A macro expanding the value-partition.
- An amino-acid function
- Intern with `intern-owl-entities`
- A highly patternized part of the ontology.

Chapter 5

Identifiers

Ontologies describe a set of entities within a domain and the relationships between them. For any practical ontology system this means that we need to be able to refer to these entities by a name, so that we can talk about the same entity in more than one place.

Deciding how to allocate and *coin* these identifiers is a key part of the ontology development process. In this chapter, we describe some of the issues with identifiers and specifically how Tawny-OWL supports a number of different approaches for allocating identifiers.

5.1 Requirements for identifiers

Identifiers in ontologies are necessary for a very simple reason; when describing the entities in our ontology, we often need to refer to a single entity in more than one place. To support this, therefore, we need a name or and identifier to refer to the entity.

OWL is the Ontology Web Language – its web nature adds an additional capability. As well as being able to refer to entities within a single ontology, we can also make references between ontologies. In the case of OWL, entities use IRIs[1]. These identifiers share a global namespace, which makes their choice more critical still.

In OWL, identifiers are not same thing as labels. A label is a human-readable string which are attached to entities while identifiers need not be.

[1]IRIs, URIs and URLs are basically all the same thing, or at least the differences between them has no practical consequences in this document.

5.2 Stability

One key requirement for identifiers

5.3 The Case for Numeric Identifiers

One solution to the problem of change management in ontologies is to use numeric identifiers. For every new concept that is created, a new numeric identifier is created. Numeric identifiers are never reused, so if a new entity is created in the future, then it wi

5.4 Identifers and Tawny

Clojure identifers, tawny names and so forth