Object Orientation in Tepal

by Howard Henson

A conceptual view of Object Oriented thinking behind the design of Tepal. Introducing conceptual OO issues and describing their implementation within the Tepal language.

Date: 07-Jan-2005

In the beginning

The Object Oriented belief

When first introduced to Object Oriented (OO) thinking, OO takes on the form of a mystical solution to solving software development issues and making the world a better place. After finally getting past the mysticism and figuring out how OO is implemented within the current languages of the day (C++ / Java) much of the enthusiasm for OO design has been stripped from the believer. Although current OO implementations provide a definite leap forward in terms of software design and development they also come with an increasing cost to solve simple problems and do not deliver against the fervour that surrounds OO development. Please note, this lack of delivery is against a belief, not that OO tools and thinking have not benefited the industry.

In order to understand the belief, it is important to follow the path to enlightenment. That is, it is important to look back and review the path taken by the OO devout in their quest to become an OO guru. The path described below is my own path.

What is an Object

Traditionally the concept of an Object and OO was presented using one of three main techniques:

1. **The taxonomy -** This approach draws on the work of biologists who have attempted to classify the living world into hierarchal taxonomies. The idea being that a class could be constructed for each categorisation of a grouping of living things and that the distinguishing characteristics are added or polymorphically overloaded on each child class producing a more and more specialised class until we get a leaf node and we create real instances from the leaf classes.
2. **The drawing program -** This is the ultimate proof that objects are the best way to write software since it is now possible to write a graphic primitive class, then extend the class to add new graphical types (such as circle, etc.). The underlying tenant is that your drawing program only needs to know how to handle generic graphical objects and you can add new objects at any point and only have to write the implementation of the new graphical object in order to extend the programs functionality.
3. **The human context -** Often classes and objects are described using the human context, i.e Human could be the class and you are the instance. Some even attempt to explain inheritance with this approach. The approach often borders on the taxonomy approach with examples of Animal - talk, Dog - talk, returns Woof, etc.

Each example is great in that it introduces some of the powerful features of the OO concept, it also high-lights some of the weaknesses, though it is not normal for the weaknesses of the analogy to be described. The weaknesses are an important factor, as such these will be investigated in this document.

What I learnt

After sitting through the above descriptions of what an Object was and receiving my first sermons on Object Orientation I held the following beliefs:

- Objects were collections of information that belonged together.
- Objects belonged together because they had behavioural characteristics that logically grouped them together.

- Inheritance was a mechanism to write less common code and to provide common properties.
- Any code could be written to work with a generic form (abstract object) and you could dynamically extend your programs by just adding new object types.

Note that the distinction between template and instance was always a little fuzzy at first. This seems to persist throughout the life time of most programmers who often incorrectly talk about the instance (or object) as the template (or class). That is, people talk about objects, when what they are actually referring to are classes. This seems to have become acceptable for general conversations, but is technically inaccurate, however at the time this was what I took out of the books and discussions at the time.

Aside: *Another interesting piece of trivia, the common belief at the time was that it was better to learn good OO principles without learning all the evil procedural coding concepts, all the good books on C++ mentioned this up front, and then proceeded to explain C++ in terms of C. Thus to someone who had no knowledge of C at the time it became impossible to learn the C++ way without first learning about the dark side.*

After going though the texts, learning about the dark side of the force (procedural C) and focussing my mind on the problem (generally with my eyes closed on the bus to University) I managed, through osmosis, to start to see the object for what it was; A truly unique and exciting piece of software magic. I wanted to do the object thing, I tried to write a graphical program, like the one used at the example to show just how simple the whole thing was. The first fracture appears within the perfection that is objects.

The magic of abstraction

I was able to identify the abstract graphic object, the point object, etc. I wrote reams of fantastic code allowing me to place the graphic object, manipulate it, etc. Now all I needed to do was add the additional graphic components and magically my program would extend itself.

It was then I realised that I could not write code that could be extended without some knowledge of all the possible graphical objects up front. The minimum requirement being to be able to list the available objects and to be able to construct new versions of these objects (this was C++ early days, and I was still a baby to the software development movement). This did not make me happy. The magic of abstraction was that I would not need to know about the leaf objects, but now my generic code relied on this knowledge. This was the first principle I noted a proper OO implementation should support:

Principle 1: *It must be possible to dynamically extend software coded against a generic abstraction without having to add knowledge of the abstractions children to the code.*

To be fair this could be done in C++ using shared libraries and a little additional code along the lines of a driver manager, but it definitely was not obvious to a novice, nor something simple to do. The idea behind the OO language is to simplify the application of good OO concepts. C++ did not do this for me. Had I have had more exposure to the state of software development at the time I may have been more fully exposed to Smalltalk, where this problem would not have been evident. Smalltalk is bound at runtime with better reflective capabilities, this would have simplified the ability to dynamically add new behaviour.

Java now also possesses this capability and makes this principle easy to implement. But still no serious focus is placed on formalising the generic ability to find leaf elements of a particular abstraction within these languages.

How objects persist

At the time of learning about all this OO stuff I was earning my spare change by writing database software using a language derived from DBase III, namely Clipper. This was a very powerful (at the time) development environment, it was compiled to native code, and allowed for advanced manipulation of what was effectively a flat file of data structured in the form of records. I was fascinated by being able to create more and more powerful data storage software. Naturally I wanted to expand this within my new OO world. I tried to find out everything there was to know about databases.

I heard of something called Oracle and SQL, this meant nothing to me at the time and since I could not put it onto my PC it meant even less. I read about how DB2 structured data (of course I still only had the vaguest idea what DB2 was). After discovering a little concerning relational databases and SQL, and my knowledge of flat file based data storage, I figured that I could take the concept of record based flat files, apply some of the concepts from my Math courses (set theory) and write an OO database. It seemed logical, SQL was not OO, therefore it was the old way of doing things, thus an OO database is the next logical step.

After a few days of work I had my first object database. This had all the glory of DB III in object form, minus indexing. I could persist my objects and read them back. Fantastic, I have stumbled on Object serialisation. Perhaps the important point here is that I learn through doing, or at least trying to do. I did not have a strong grasp on data storage at the time, nor did I fully understand OO. I definitely did not understand the truth of relational databases, but I did have an intuitive feeling for these things.

After my initial success I figured I need to learn more. This lead to a study of every book I could find about how to write object databases. As it turned out this was quite a quick study as the only work I could find in the library was Dr Stonebreaks work on Postgress. I did not fully understand it.

I searched the Internet, a new discovery (the Internet at the time was a resource reserved for the rich, or special university students. I had only just managed to approach the level of a student who wanted to be special but managed to get access to the Internet anyway), I found all kinds of work. This tied with my new appreciation for a little know operating system called Linux allowed me to compile and play with work from many other people who had also been grappling with this problem. All the work I reviewed displayed some great ideas, but none of them engendering the feeling of being correct (that is what it was I was looking for).

By now I felt that I had become quite proficient in C++ coding and naturally thought that meant I knew all there was to know about OO, after all C++ was object oriented, I could program effectively using C++, therefore I understood OO.

It was around this time I stumbled into a Lecturer who had similar interests, he was writing up his doctoral thesis on Object Oriented Databases. This was amazing, he gave me some very interesting articles and books on Object Orientation that I probably would never have found otherwise. He also asked me questions, to which, in my arrogance, I had all the answers to. It was these questions that ultimately forced me to ponder on the true nature of object orientation.

The question that stands out in my mind still today was, “Given an Object car, how do you implement drive?”. This seemed obvious, I started rattling off a solution. Then he started asking other probing questions about the solution, such as, “How does drive relate to the

object steering wheel, how do all the object co-operate, where does the drive message get satisfied?". I finally stopped talking and started thinking.

I soon realised that there was a lot more to this object thing than my simplistic initial understanding. After trying to understand object data storage I started to see that object storage should be a characteristic of the object. If we were to provide true object storage it would be imperative that the behaviour and data should be stored (and retrieved) as a whole, and not separated as in many of the other solutions I had encountered. In order to achieve this it seemed evident that the code aspect of the object would need to be transportable across platforms. C++ did not fulfil this need, not even sort-of. If I retrieved an object from the database I would need to be able to execute the object, this would require all clients and the server to be able to execute the same code. Since it was unlikely that all clients and servers would run the same operating system and have the same processor the objects executable format would need to be transportable.

Principle 2 - *Object code needs to be transportable across both operating system as well as underlying hardware.*

I racked by brains thinking about how this could be achieved, I had considered using .obj files since they were in a format that should be neutral, then using a linker on the target platform on demand. This seemed like more work then I was willing to put into this pursuit. I stopped worrying about object databases for a while (If only I had known about Smalltalk).

A few years later Java made its debut, with it's write-once run-anywhere marketing I found new interest in Java as a language capable of supporting the needs of an OODB.

CORBA

Whilst worrying about how to move objects from one computer to another I stumbled on CORBA. CORBA looked like the magic I had been looking for, the ability to run an object on a computer that was not the computer holding the object. At first I interpreted this to be the ability to serve an object to a client and allow the client to execute the object. This would tie up with my need to run object code anywhere. Sadly this was not the case, however, it did claim to allow a computer to execute methods on an object remotely. An interesting approach as this could also be a manor of dealing with my object database.

To cut a long story short, I ultimately found that CORBA was another way of saying remote procedure call. It's ability to deal with objects was extremely limited and did not support the concept of migrating object from one computer to another, nor truly support the arbitrary remote execution of methods on objects. All in all a very disappointing experience, one I have been forced to re-live with EJB and to a lesser extend Java's RMI.

The magic I could not find can be encapsulated as:

Principle 3 - *Objects should support migration over a network as well as support the transparent consumption of messages over a networked environment. Where an object resides should be a matter of deployment and tuning, not programming.*

Messages

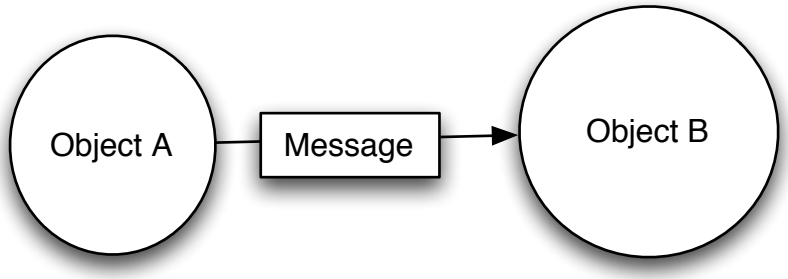
Our understanding of the world we live in expands little by little, building on past experiences and knowledge gained. Our knowledge is gained by grasping at pieces and slowly fleshing it out. Even if you were given everything you would ever need to know, you can only hold on a little at a time, loosing most of the information like sand through a sieve. In order to gain knowledge, we grasp at whatever we can, weaving our prize into our net of

knowledge and continually trying to keep adding to it. I have encountered the terminology of message parsing a number of times as I have build my net of knowledge about Objects.

I first marvelled at the magic of being able to send messages to object, and then letting the object decide how to deal with the message, as described when learning about Smalltalk for the first time. Since I had no access to any useful Smalltalk environment at the time I looked to C++ to provide the framework to understand this concept, and discovered messages were function calls, and the contents of the messages just method arguments. This was deeply disappointing, I had visions of this unstructured mechanism of communicating with an unknown object and it magically being able to handle (or not handle) the message. This was dynamic and seemed to be a powerful way of supporting my first principle in that I could just send messages to objects and it would just respond. I could magically send create messages, etc. I have re-thought about this many times, the most notable time was when I worked with Message Oriented Middleware. This is where I could see structure to messages and a mechanism to deliver and use messages.

This lead to the next concept:

Principle 4 - *Objects should pass messages to each other in order to support inter-object communication. Messages should be implemented to support dynamic handling and/or routing of messages. Messages passing should support run-time binding, that is compile time checking of message parsing should be optionally supported. Messages should be self contained and be able to pass over networks in order to be delivered to the appropriate object.*



This is perhaps one of the most vibrant concepts I had mystified with respect to supporting the object oriented paradigm. Loose coupling is always implied through the concepts of abstraction, and formalised during my initial exposure to Smalltalk.

Memory and Storage

When starting out with C++ memory management is also very interesting. How are objects represented in memory, you need to manage memory, but not to much. Later in my quest to persist objects the internal structure of the object becomes more and more important. There are concepts such as in-direct pointers, swizzleing, and virtual memory.

In C++ there is a lot of focus on memory management, with very little support. In Java there is also a lot of focus on memory management, but in this context, the focus is in ensuring that do don't (manage memory). Both approaches have there strengths. Java leaves a lot up to the environment, which for most normal programming jobs is perfect, but if you are writing an OODB it makes things just that little more difficult as no standard hooks exist to manipulate the memory foot print of the objects within the context of the language. Some OODB's such as Gemstone delve directly into the JVM replacing the

memory management routines, a noble approach, but one fraught with many issues, especially when interacting with the hot spot compiler.

It quickly become apparent though that the memory representation of an object should not be constrained to a particular media. That is an object should be able to be hosted as bits in main memory, bits on a disk, bits in a stream. The only difference is the speed at which it can be worked with. Obviously the current hardware requires the object code to be in main memory in order to execute the code and use the data of an object, but that should be transparent to the developer.

Principle 5 - *The representation of an object should be independent of the physical location of the representation, i.e. The programmer should not be concerned if the object is currently located in memory, the hard-disk, or some other storage media. The movement of the object from one media source to another should be transparent to the programmer and a matter that could be configured / tuned by the deployer or administrator of the deployed application.*

This statement is not the same as serialisation. With serialisation the object is cloned into a format that can be used to 're-instate' the object at a later time, however the re-instated object is not the same as the original object, that is for object *a* and the re-instated version of *a* (say *a'*) does not support *a == a'*. In order to support principle 5 the mechanism should ensure that *a == a'* and also ensure that only one correct copy of *a* exists (or master copy).

How to drive a car

Previously I mentioned the car problem and how to implement drive. This problem baffled me for a long time. Not that I couldn't find solutions to the problem. Just that none of them felt I had successfully captured the essence of the problem. As it turns out, with all the work I have done designing object databases and the like, I believe I have finally found an answer that works for me.

The issue is with Object categorisation. Because of the hierarchal mind set burnt into my mind from the taxonomy explanation of objects and that languages generally provide one mechanism to template from we loose potential detail.

Typically we model entities, for example in the car scenario we may model wheels, steering wheel, pedals, engine, etc. We then model the car as an aggregate of the components, four wheels, one steering-wheel, three pedals, etc. Finally we attempt to associate behaviour with the components. My first thoughts on the drive problem went along the lines, car supports drive behaviour. Then drive uses steering-wheel, etc. Although we may be able to capture some of the essence of driving a car, we haven't correctly modelled drive since the work we have done does not support re-use as in real life we would be able to re-use our ability to drive a car to driving other vehicles. We would need to re-implement a lot of the concepts in the other vehicle construction.

Java introduced the notion of different types of templates, classes and interfaces. Classes were the cookie cutters providing implementations, Interfaces providing common behavioural templates ensuring a common standard for naming and identifying when a behaviour had been captured within a class. This got me thinking. Perhaps we have over simplified the problem to something so generic that we loose the wealth of implicate knowledge that could be contained in a more rich collection of templating devices. As Java had intro-

duced the concept of Interfaces, perhaps we could apply this approach to solve the driving problem[1].

It is important to first recognise that some types of objects represent things. Things could be defined as objects that have a tangible representation and contain properties that are constant over time. We can call this class of objects entity objects and construct the object using the entity keyword, i.e.

```
entity Wheel.
```

Another class of object type could be relationship or role. This type of object associates one entity object with another and may contain additional information that exists as a result of the association. This could be realised using the keyword role, e.g.

```
role CarWheel appliesTo Wheel belongsTo Car
```

This describes the relationship between the Car and it's Wheel object. The Wheel becomes part of the car and for the duration of the relationship it belongs to the Car. In this case car acts as both an entity as well as a collection. The Car and its core attributes define the Car even when the wheel is changed. The Wheel also exists independently of the Car, when the Wheel is removed from Car the Wheel still exists, just under a new association.

The CarWheel object can have new properties that were not relevant when the Wheel was on its' own, for example it is now possible to have a torqueApplied message slot to support the drive method.

This allows us to describe transient associations, but still does not answer the drive question. So the next step is to look at the object, with all its associations and consider how we think about a car as a driver. As a driver we do not care about the engine, wheels, etc. Nor do we care about how they are all related. We only care about the abstractions we use to control the vehicle. This leads to to the view or filter constraint. We apply an abstraction to the Car abstraction to simplify and associate additional knowledge to support driving the car. The filter is likely to provide an abstraction relevant to other forms of transport as well. In this case we create the view called DriverControls, this could be done using a keyword called view, for example:

```
view CarDriverControls inheritsFrom DriverControls appliesTo Car
```

It would then be possible to provide a set of meaningful abstractions to the view, such as turn, accelerate, break, etc. This could then be used by the role object associated to the entity Person, namely the role CarDriver (which inherits from Driver). To which the message slot drive could finally be associated.

It is useful to note that a role defined with appliesTo dynamically extends the entity object adding the roles behaviours and properties to the entity object in question. Thus in this case the Person entity object would now be endowed with the behaviours of CarDriver, thus the message drive could be sent to the Person object in question and the person would then drive the car included as part of the message.

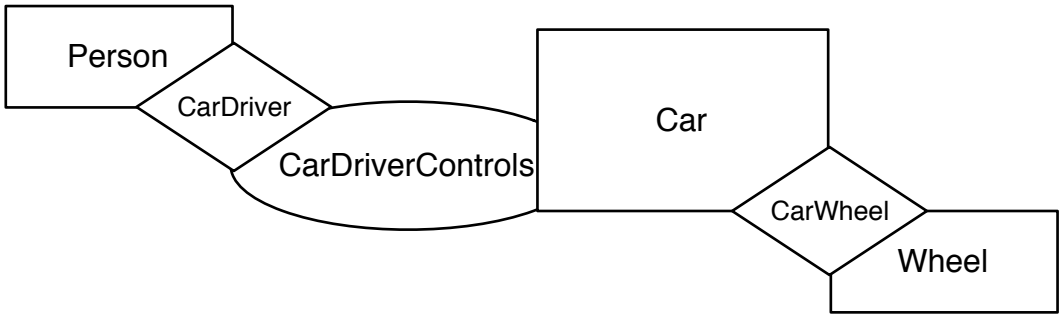
Principle 6 - *The language must support meaningful keywords to simplify and standardise the approach to implementing common design approaches. That is, a single class keyword is not sufficient to abstract and group the contextual use of the objects.*

---

[1] There is no one solution to this problem and it can be solved in a more generic way using the current standard languages, including assembler. The issue is not that the problem can be solved, but rather the support and guidance the language provides to simplify or standardize the approach to solving the problem.

The corollary to this that the language must support a pattern based approach to supporting the construction of class templates that is realised using a well defined set of keywords denoting the pattern of the class template to produce.

A picture may be helpful, the diagram below shows a stylised representation of the different elements and their associations.



Another important point highlighted previously is that an object does not remain static throughout time and should be able to dynamically add and remove characteristic throughout the lifetime of the object. For example, a Person does not start its existence as a CarDriver, however in time this role is applied and may at any point before the end of the objects existence be removed. The role has the keyword appliesTo which advises the environment to which object the role is designed to support. When the role is actually applied the instance object to which the role is applied is changed to implement the role (using Java terminology). Only the instances which are associated to the role are changed, thus if this were Java not all cases of Person instanceof CarDriver would return true. However, unlike Java it is possible that the following could be true (using a Java style syntax):

```
Person person = new Person();
person instanceof CarDriver; //returns false
CarDriver carDriver = new CarDriver(person);
person instanceof CarDriver; //returns true
```

Principle 7 - *The type of an object instance is not immutable after object creation. It is possible to extend (or restrict) the behaviour of the object at runtime. This should be managed in a controlled manor that ensures the potential for type safety.*

Broken Analogies

As pointed out in the beginning of this document the analogies used to explain OO also have their problems. The taxonomy approach, this uses a single inheritance mechanism to partition all living things. In real life the taxonomy approach has a hard time classifying life as it fails to deal with cross cutting concerns. That is, at leaf nodes, not all elements fully satisfy the partitioning that they are allocated to. Also there are shared characteristics that are found in multiple elements of a grouping that are found in other groupings, but do not justify a separate grouping as the other dominant characteristics direct the placing of the element in a the grouping they are currently allocated to.

This is a similar problem found in languages that support single inheritance. In languages where multiple inheritance is supported it is possible to allocate multiple groupings to an element. Unfortunately this ability also causes significant issues (note the fragile base class issue). However, the ability to support partial characteristics that can be applied to a class in a manor similar to that of multiple inheritance is quite useful. The role and view

types go a long way to allocate generic characteristics to an object that do not necessarily follow the same inheritance path as the entity object. However, there is still more than can be done.

A lot of work has been done in this space using Aspects to extend and dynamically change the behaviour and nature of objects. Aspects can provide a structured mechanism to change behaviour of objects. The underlying requirement though is the ability to reflect and modify behaviour as well as characteristics of objects. The exact mechanism required is not as important.

Principle 8 - *The behaviour and characteristics of an object should support dynamic reflection and introspection.*

This principle may have to be controlled as it would have interesting side effects for commercial code where the producers of the code may not wish to expose the implementation of the behaviour, where introspection of the code would allow for de-compilation of the code to a level where the behavioural mechanism would be exposed.

Needful things

We have covered some of the perceptions of OO and the positive misconceptions that were produced. This section focuses on the aspects of an OO system that need to exist in order to be functional and support the development of real systems.

Persistence re-visited

Many of today's existing and legacy systems make use of persistence mechanisms, these include flat files, as well as relational databases. Most modern systems make exclusive use of relational databases to store their information.

This is done not because of the purity of the solution, and definitely not because of the close coupling between OO design and RDBMS design. Rather this is done for the following justifications:

- **Legacy integration -** Legacy systems exist that require integration and it is felt that this would be eased if the new system stored its data in a similar technology.
- **Tool support -** There are a lot of tools available for RDBMS based solutions, thus by developing against this 'standard' it is possible to leverage the work done in this space.
- **Available Knowledge -** Lots of people know relational databases / SQL, thus it is better to use a relational database to store the information.
- **The Lemming excuse -** Everyone else does it this way, so why should we be different. This is the same as the 'Why re-invent the wheel' excuse.

These are the main justifications I have encountered when questioning the use of an RDBMs for data storage in OO world. Other more plausible reasons include performance, stability, product maturity.

After spending a large amount of time working with both relational and object database storage solutions I have come to the following conclusions. Both approaches have their good and bad points. It is my opinion though that it does not make sense to pollute the RDBMS environment with OO wrappers, if an RDBMS is chosen for storing data, then the appropriate design approach should be taken. ER modelling is a form of object modelling and with sufficient care the relational model can produce good quality property abstraction. This is likely to be very similar to the object model that may be produced should the solution have been modelled using an OO approach, but it would not be exactly the same, nor should it be.

The common industry mistake is to take a clean data-model and hide it under layer upon layer of object abstractions, where either the object abstractions are little more than a tacky veneer on the ER model, or a brilliant, but mismatched, re-abstraction of the problem space. In either case the layer produces additional complexity, that contrary to popular opinion does not improve the quality of the solution.

Each layer adds a performance overhead, moves the user one step away from the true representation of the information and looses context. A typical J2EE solution would attempt to wrap the database in one of several popular techniques:

- **The Object-Relational mapping layer** - There are two alternative approaches to this, only one is useful. The first tries to map relation characteristics to an OO environment, the other attempts to map OO characteristics to the relation environment. When the later is used and the relation design is generated from the object design, then the RDBMS becomes a glorified structured file system. This allows for limited OO design to be successfully implemented, but likely at the loss of ease of use by third party tools (often sited as the justification to use a relational data store). The former solution either creates objects that are poorly designed (or rather not designed) or solutions with extreme mismatch producing extremely poor performance of the persistence solution. Either way the former is not a reasonable solution.
- **The CMP/BMP -** This has all the problems of the mapping layer, without the performance nor the grace. The design of this solution is a knee jerk response to beating Microsoft on features and should never have come to life. Much work has been put into trying to breath life into this standard, however some things are better lost.
- **JDO -** So called standard object database support layer that allows for supporting object database and relation mapping tools with one common API that will allow the whole world to work together. Sounds interesting. The API is heavy, extremely relational mapping focused and its ability to support a true object database is only relevant so long as the database looks like an object relational database (i.e. Poet / Versant), but will never be able to work seamlessly with a real object database due to its relational background.

Probably the worst side effect of entertaining this layered ‘de-coupling’ of data representations I have witnessed is the lack of understanding it engenders. Since the developers are exposed to a Java object interface to the database, they never get to understand the design of the underlying database, as such they start to code their software as though they were working with actual objects and produce high tuned code similar to the example below:

```
List results = Accounts.getAllForClient(“Mr X”);
double outstandingBalance = 0.0;
for( Iterator elements = results.iterator(); elements.hasNext();){
  Item item = (Item) elements.next();
  if( item.isUpaid() ){
    outstandingBalance += item.value();
  }
}
return outstandingBalance;
```

On the surface this looks like well written code, the meaning is clear, the appropriate collections are used, etc. The issue is that the method getAllForClient requires the execution of SQL code against the database, the results are then returned and are then re-queried in the Java code. This requires the de-serialisation of the JDBC result stream, and the creation and population of the Item object from the result stream in order to query a property

and count another based on the query result. Had this been coded in SQL the query may have appeared something like this:

```
SELECT SUM(i.value) FROM items i
  WHERE i.accountName = 'Mr X' and i.paid = FALSE
```

The code is more compact, it performs the same task, all the logic is executed at the source of the data, the de-serialisation requirements would be limited to a single double value, and the database can be used to do what it is designed to do.

A common justification for the use of multiple layers in enterprise architectures is the belief that by creating an abstraction to the underlying data the code in the layers above can be protected from change to the underlying layers properties. This belief has its roots in traditional middle-ware solutions such as message oriented middle-ware where the message abstraction is capable of providing a meaningful business view on the underlying processes and in theory was able to allow multiple systems with different representations of the same business entities to communicate (c.f. SWIFT). When the correct context is applied, i.e. multiple disparate environments with their own code and data representations it is a very useful architectural tool to create a neutral business representation of the systems that is capable of protecting the systems above the layer from the complexities of the systems below the layer. However, this approach has become so intrenched in the texts and minds of the current generation of software practitioners, to the point where the context for the appropriate application of the tool is lost and all solutions now follow this pattern of layering (of data model representations). This has produce overly complex solutions increasing the cost of the development and maintenance of the resultant solutions instead of reducing the cost as the initial usage of the architectural tool was designed to do.

When a solution is responsible for its own data representation and does not require the ability to make use of multiple different data stores of the same (in type) information, then the data representation should match the business data model describing the solution. That is to say, should the solution be modelled to make use of an OO design, this design should be the same as the persistence design. A tool like Hibernate could be used to perform this task, however, unless the use of a relational database it is absolutely essential, avoid the use of traditional relational databases.

Principle 9 - *The Object runtime environment must be able to support the persistence of data objects. The languages standard libraries should include collection types to facilitate the efficient storage and retrieval of persistent objects. The ability to retrieve objects from the collections should not require the introduction of any additional languages (e.g. should not require OQL to extract objects). The solution should provide an appropriate API and structure to write extraction logic in the native language syntax.*

By storing the objects in the same form they are represented in it is now possible to write correct code similar to that presented in the above Java example, where the language used to solve the business problem does not have to be diluted with the use of SQL or other non-object based query languages, whilst not creating the performance problems as would be in the scenario presented above.

An example of how this may appear is presented below:

```
account Account := { account in Accounts | account.owner.name = 'Mr X' }
result :=[count, value] -> { item in account | item.unpaid = false }
```

In the example above the *{}* indicate a set selection operation. The result is either a single element (in this case it is assumed that a single element results in the element and not the set of the element) or in a set of result elements. the clause after the | symbol indicates the

condition that should be applied in order to generate the set, and the *in* operator advises that the item to the right is the super set of elements and the item on the left of the operator is the name of the element that is used in the condition statement.

The message *count*, with parameter *value*, that is the property value is passed to the set formed by the item selection, and the result of the message is forwarded to the result keyword.

In this case all elements are consistent with the language and no additional query tools were required. The compiler could choose how best to optimise the query, potentially by delaying the set creation until the set is actually required and then only creating the set sufficiently to support the needs of the message passed to it.

The network is the computer

The phrase, 'The network is the computer' has been bandied about, to me I take this to mean that the use of a network should not be any more complicated or different to doing things on a single computer. That is to say, from a programmers perspective there should not be any significant difference to developing software accessed over a network versus using that software on a single computer. The actual state of play, although much better then it has been in the past, does not match this understanding. Standards such as CORBA and subsequently EJB (for Java) have moved this dream forward, though only incrementally. Work done on an open source project called Avalon, more specifically the AltRMI work moved this forward even more spectacularly. Unfortunately though there is still a gulf between the complexity of writing software for a single computer and writing software that works on multiple computers connected via a network.

Given the level of experience and history with networking it would seem that a language should come forward that would support networking transparently (as with persistence) allowing for software to be executed with no additional software development effort. Some prototype work in this space is available in the form of JavaParty (another open-source development effort).

Principle 10 - *The OO environment should support transparent execution of code across network infrastructure. Further to this, the standard libraries to access and deal with resources should not provide a different implementation for dealing with network resources than for those dealing with file or in memory resources.*

Smalltalk

Smalltalk has been mentioned a number of times, this language developed before C++ and Java, is a language before its time. The concepts pioneered in this language are still at times more pure then that found in Java, which drew on the strengths of Smalltalk and Eiffel. Some of the core concepts that deserve special attention include:

- **Everything is an object -** Literally every aspect of Smalltak is implemented as an object. This includes what is commonly referred to as primitive types (int, char, etc.).
- **The virtual machine -** By abstracting the runtime environment from the 'compiled code' it is easier to port the environment.
- **Dynamic as well as compiled execution -** That is it is possible to execute code dynamically within the running environment without requiring a formal compile cycle.
- **Runtime binding / Late binding -** The binding of the message send to the object is performed at run time.

A good language should be consistent with its metaphors. Smalltalk takes this to the extreme at the grammar level. An OO language should not provide non-object metaphors

within itself. This is the case with Java, where primitive types have a different status and method of handling that is not the same as for any other object.

The manor in which the compiler produces the code is not relevant, thus if a compiler detects primitive types, there is no reason for it not to reduce the code to the equivalent of primitive types, but at the language level this should be totally transparent.

Principle 11 - *The language should be consistent. An effect of this is that everything should be viewed as an object in the language, that is there are no primitive types. This also touches on the nature of a message and the 'class templates'. The general rule should be if you can reference it, it is and object. Messages should be referable, that is it should be possible to create a message and refer to the message via a variable reference. Thus a message is an object.*

The concept of a virtual machine to which all code is compiled is a powerful mechanism to support platform independence allowing code to view a consistent byte code format that all tools can target. It should however, be possible to perform an additional compilation / linking to convert the platform neutral byte code to native platform specific code. The process should be light weight and be able to be performed on the fly (aka jit compiler).

The advantage of compiling the code to native machine instructions is obvious. Although there is real advantage in being able to tune the compilation using execution statistics, it should also be possible to perform a first pass conversion without the statistics. This would help any long running code. It should also be possible to re-compile the code using execution statistics gathered during real use of the code. This would allow for creating model use scenarios to pre-tune the compilation process (could be downloaded with the byte-code) as well as for ongoing tuning based on actual utilisation.

Principle 12 - *The template defining an object and its behaviour should be represented in a platform neutral manor. The execution environment should support the execution of this native format as well as providing support for dynamic and potentially static conversion of this format to the underlying platforms native execution format. When template code is transported (i.e. over a network resource, stored onto a hard-disk, etc.), the smallest transfer representation should include the platform neutral representation. It is allowable to hold additional representation formats where applicable, for example when storing templates to a persistent storage device it may be useful to store the native binary representation of the template as well.*

A dynamic language should not be limited to compile only execution, in that it should not be required to compile source as a separate process to the execution of the code. It should be possible to parse code at runtime and execute the code within the execution engine. This may be implemented internally as a code compilation, but the user effect should be that dynamic code could be constructed and executed within the runtime environment.

Principle 13 - *The language should support dynamic execution of scripted elements. That is, the executing code should be able to introduce dynamically constructed source into the execution environment and be able to execute the newly introduced source code.*

How to do it

The previous section presented many concepts that need to be associated to an OO language, as well as highlighting how other languages do not support these concepts. This section describes how the principles already identified may be applied within a language syntax, as well as introducing the syntax of Tepal.

Source Code

Possibly the best place to start is with the source code used to define application behaviour. The most fundamental aspect is how the source code is represented. There are many alternatives, with the most popular to store the code in plain text files. In the short term this is a convenient mechanism that will allow us to rapidly prototype the new language without complicating the environment with needing to develop custom editors.

The use of a code database would be more consistent with the language objectives (in that the language supports transparent persistence of all objects). The use of a database for code storage would allow for more powerful revision control and allow for tracking additional meta-data such as re-factor operations that could be used to support object migration of persistent stores. As the language matures it should move from flat file representations of code to more rich persistent source objects.

File structure

The initial structure for the source language should follow the same approach as that for Java. That is each file should contain one type description. Each file will contain the complete description for the type within the file (i.e. **not** like C/C++).

The files are organised into logical structures and the structures are represented by organising the files within directories with the same naming structure of the groupings (i.e. as in Java).

Although this strategy has a number of weaknesses and limitations, the approach is generally useful and helps manage the complexity of grouping and finding related types as well as been comfortable to use.

The text file should be named with the name of the type and have the extension .tepal appended in order to identify files containing Tepal code.

Comments

Code commenting is generally agreed as being very important in order to assist with the explanation of why an approach is taken or how the code is believed to work. Recently code comments also include documentation describing the use of type been defined or method, etc.

In Tepal code comments will take the standard Java syntax. Once the code is coded within a formal Tepal IDE, and the code is persisted as objects, comments can be associated using a comment embedding approach where comments can be embedded within the source object. However, in the short term using the Java conventions should be sufficient.

The code comment types are:

```
//The double forward slash comment or line comment.
//This comment indicates that the remainder of the line is a comment
//Once a new line is detected then the comment is over.
```

```
/* The comment block, this is useful to block out a section of code
  an all the enclosed elements are deemed as comment. This is
  most useful for larger comments or for temporarily removing
  code elements.
*/
```

The other type of comment type is the code documentation comment. These structures contain documentation describing the use of a property, message, or type. Once again a short term approach is required in order to support this need. The use of code documentation is very helpful to support the effective use of types, etc. However, a better mechanism is required to provide more complete support for code documentation. Once again the approach taken by Java will be loosely followed. The approach is depicted below:

```
/**
 * The use of the double star and the repeated * on each line is
 * the differentiating characteristic.
 */
```

Further descriptions of code documentation will be made as appropriate as the ability to document code is extremely important.

General code syntax

The general syntax for defining a type is described as follows:

```
type MyType further type attributes:
    section:
        section contents
        sub-section:
            sub-section contents
            more sub-section contents
        ;
    ;
;
```

A few patterns are easily identified. The first is that the basic block structure is identified by the ‘:’ character, then the ‘;’ character terminates the block. Originally I had considered just using indentation to denote blocks, but after thinking about it I was reminded about the difficulties of the Make files that have white space sensitivity, as such it seemed prudent to make use of a more formal block marker. The choice of the : and ; is a factor of prettiness, the blocks could also be denoted by the more traditional ‘{‘ and ‘}’ characters, though I would like to reserve those characters for describing sets in keeping with standard mathematical notations.

The statement terminator is the carriage return / new line character. This is a loose definition as it is possible to split a statement over multiple lines, however when the first element of the next line is no longer considered part of the current statement, then the statement is assumed to be terminated.

The *type* element is a place holder for the type of the template been constructed. Examples of *type* elements could be entity, role, etc. The type names are in camel case with the first letter being lower case. The name of the template follows the type element and must be unique within its name space and is formatted as camel case with the first letter being capitalised. The name space is identified via its location within a directory. The name space is the names of the directories from the source root, concatenated with a ‘.’ character to the current source file location. This is similar to the Java approach to packages.

Each type is capable of dealing with additional attributes that are used to convey further information to the type and is used to describe the configuration of the type. Consider the type as being an object, the type is constructed with the ‘parameters’ following the type keyword.

It is possible to place the *abstract* keyword before the type declaration. This indicates the type is a partial type and should be merged with the non-abstract child type. This can only be used with types that support the *inherits* keyword in its’ attributes.

All keywords used in type attributes have well defined meanings and types requiring the functionality associated to itself should re-use already defined keywords to ensure consistency.

Message passing

The following syntax shows an example of message passing:

```
[messageId, parameter1, ...] -> anObject
```

The ‘[‘ and ‘]’ characters are used to identify the beginning and end of a message. The first element of a message is the messageId. The message id should be camel case, with the first letter being in lower case. The message id denotes the action the message is intended to trigger. The message then contains a number of optional parameters that make up the message body. The parameters are either an in order list, with missing parameters indicated by not placing any value between the comma’s separating the list, e.g.

```
[myAction, p1, ,p2]
```

Any parameters not added to the end of the list are automatically assumed to be missing.

Messages can also be constructed using key-value pairs. Each parameter has a key associated to it. This is highlighted below:

```
[myAction, key1:p1, key6:p6, key3:p3]
```

As is shown the key is placed before the parameter and separated by the ‘:’ character. Also the keys can appear in any order, missing keys implies missing values. There is a difference between a missing value and an empty value. It is possible that the supplied value may be empty, but the user supplied the value, since every thing is an object it is possible to change the value of non-constant objects, whereas if no value was supplied then there is no reference to change.

It is also possible to work with a message as an object, for example:

```
var1 := [myAction, p1,p2]
var2 := [myAction]
var2 += key1:p1
var2 += var1.key2
```

In these examples we see a variable been assigned a message. In the first example the message has a number of parameters assigned to it, the second only defines the message type or message id. It is also possible to modify the message once it is constructed. This is displayed by adding the parameter p1 at position key1 to the message. Finally it is possible to interrogate the values of a message. This is done by using the ‘.’ character to de-reference the object and the property to de-reference. Each key becomes a property in the message and is able to refer to the key-value pair.

The message id itself can be referred to via the property ‘type’. This is used to denote the object relationship in that a message is characterised generically by the message id and

that instances of this generic are possible, i.e. the message id can be considered a type, thus the name of the type is the message id.

Along a similar line it is very important to ensure that message types are re-used where possible, that is an object that is capable of performing a specific task similar to another object should use the same basic message syntax and structure to perform the task. The net result should be a significant reduction in the number of message types. A counter example could be found in Java's collection API, consider:

```
Map map;
map.put( "Key", "Value" );
Set set;
set.add( "Value" );
```

Both methods deal with adding an element to the collection, but different messages are used to perform the task. In Tepal this may take the form:

```
[add, key:"Key", value:"Value"] -> map
[add, value:"Value"] -> set
```

The message structure here repeats the notion of addition, in this case set addition, of which a map is just a special type of set which can take an additional parameter, namely a key value. This simple approach can be extended to messages Java would consider as keywords, such as *new*. There is no reason for this to be a keyword, but rather a message we can send to a template in order to create a new object from the pattern it describes, for example:

```
map := [new, initialCapacity:11 ] -> Map
set := [new] -> Set
```

In this case the Map and Set templates are passed the message *new*, the result of which is captured into the map and set variables respectively. Message results can be chained, that is, it is possible for the result of a message to be a message. It is also possible to embed messages within messages, for example:

```
[add, key:"MyNumber", value:[new, precision:5]->Number]->map
```

This could be interpreted as sending the message *add* to the map with a new number of precision 5 as its value.

Order of execution

A standard feature of most languages is the ability to predict (within the bounds of a single thread) the order of execution of statements. Typically the first statement is executed and then the next and so on. Tepal has no such concept. Statements are not guaranteed to execute in the order they appear. This may sound confusing, it probably will be. There is value in this though, the value is that the execution of statements can be parallelised, that is the execution engine is able to re-order statements to optimise the execution of the statements across multiple processors and/or computers. The runtime engine will attempt to determine the join-points of a statement list, or the points at which different streams of code execution are required to merge. This is typically identified through the dependency on a variable result.

In order to try and explain how this may occur consider the following simple example:

```
var := [new ] -> Set
var2 := a + b
[add, var2]->var
```

In this example there are 3 visible statements. The creation of 2 variables and the use of one variable as a message value to send to another. A simplistic runtime engine may choose to create the new set, then store a reference to the construction code for var2 and delay the construction of var2 untill it is actually consumed by the set. Another approach may be to establish two treads, one thread given the task of creating the new set and the other the task of performing the addition. Then finally merging the threads and performing construction of the message and send the message to the var object for processing.

The idea is that each code segment be given a cost and be grouped by the compiler for parallel execution allowing the runtime engine to perform a cost based analysis for the most appropriate execution of the code groups. The upside is the potential to make best use of multi-core / smp computer resources as well as being able to support distributed code execution without forcing the programmer to make the decision about how best to distribute the load.

This brings us to one thing that Tepal does not do, that is threads. Tepal has no programmer notion of threading. Or rather their are no direct mechanisms to initiate threads of execution. The construction and management of threads, as well as the distribution of processing amongst threads is the responsibility of the execution environment, with support from the compiler (see the section on Threads for more information).

In order to make best use of this approach it is important that information about what is required of the program is best described in a manor where dependencies are easy to identify.

There are three main concepts that help support this approach to execution:

- **Declarative programming -** As far as possible information is declared to the system, the system is then responsible for interpreting the information into an implementation strategy.
- **Event driven programming -** Code that is event based makes no assumptions as to the path taken to reach the point of execution. It is also relevant to indicate the conditions which trigger the events.
- **Rule based execution -** Using a rules system to indicate pre-conditions to the execution of actions or events within the object(s) allows for a formal means of triggering events.

The type mechanism used in Tepal strongly supports the ability to declare high level information that can be interpreted by the type implementation (the implementation of the keyword, not the type constructed using the keyword). The type becomes responsible for identifying the strategy and for providing an implementation to the strategy.

An example of this is presented below:

```
entity Person:
    properties:
        name String
        dateOfBirth Date
        dateOfDeath Date
    ;
    constraints:
        dobOrder:
            dateOfDeath > dateOfBirth
        ;
    ;
;
```

In this example the entity type is responsible for providing a strategy for the construction of the declared properties, as well as associating the constraints supplied to the appropriate event rules. Once the rule is declared the programmer is guaranteed that the rule will always be maintained. This is a very simple example. The use of declarative programming is not restricted to types, but is the responsibility of all programmers to ensure that pattern is applied consistently.

Note that the event mechanism is used to determine when to call the constraint rule. In Java this would take a much larger amount of code to implement and is likely to more complex to change the rule.

NOTE: More work is required to explain how this works, as well as providing a more detailed set of definitions for rule based code definitions.

Operators

Operators are not messages, they cannot be referred to as a variable. They only exists within the context that they are found and are used to transform the surrounding objects based on the pattern described by the operators. Operators are not limited to system defined implementations and the implementations of the operators can be overridden or changed.

Some basic operators are defined below:

- **Assignment ‘:=’ -** This operator is interpreted as associating the expression to the right of the operator as being the value of the reference to the left. If the reference does not exist a new reference is constructed as a temporary variable of type defined by the result of the expression. The evaluation of the expression to the right is not guaranteed to occur until the reference to the left is de-referenced or in the case of a non-variable reference, the expression is going to go out of scope. Assignment is system defined for all types and should not be modified. It may be useful to leave open a small gap for special cases.
- **Addition ‘+’** - The additive operator is interpreted to produce a result of the combination of left value to the right value such that the left and right values could be factored out of the result.
- **Additive Assignment ‘+=’ -** This is similar to the standard additive operator, except that instead of producing a result, the left value is added to.
- **Multiplication ‘*’ -** Produces a result by combining the left and right values similar to addition, but with the underlying association as per numeric multiplication.

The list continues, but suffice to say that the pattern is the same. There are a number of classes of operator, namely:

- **Mathematical -** Such as *plus* and *minus*
- **Logical -** Such as *and* and *or*
- **Associative -** Brackets, ordering
- **Manipulative -** Assignment, dereferencing, array references

In theory all operators can have there meaning modified and manipulated, however this should as always be done with care. Some operators are more useful to implement or override then others, for example it is useful to provide ordering (<, >) implementations, but not generally useful to provide and implementation for brackets (‘(‘, ‘)’).

The rules defining precedence are not modifiable.

The set of operators should be able to be extended to support new operators, so for example if the set template requires the special additive operator to define a different form of operator it may create the '(+)' operator.

Another type of operator is the scope operator. The scope operator is responsible for providing a hint to the compiler / run time engine as to the expected type of an object. This is useful when attempting to resolve which operator instance to make use of. An example of the scope operator is shown below:

```
aVal := (Date)"1 Jan 2005"
```

When resolving an operator it is important to understand the rules that are used to describe which operator to use. For example:

```
aVal := 1 + "2"
```

What rules determine the approach to evaluating this expression? One approach may be to convert the 1 to a string "1" and then evaluating the expression as a string concatenation producing the string "12", the other approach may be to convert the string "2" to the number 2 and then interpreting the operation as a mathematical add of 2 numbers resulting in the number 3.

In Tepal the evaluation of a statement will:

1.Try not to cast the expression, thus if both left and right values of the current operation are of the same type, it will not attempt to convert the types.

2.In the case of type mismatch Tepal will attempt to convert the type to that of the receiving variable if the place holder has a specific type.

3.If the assignment type is undefined then Tepal attempts to convert to left type.

4.If there is no conversion to left type then Tepal attempts to convert to right type.

5.Should neither conversions prove successful, Tepal will cause an exception to be noted and fail the processing of the expression.

The cast operator allows for programmer intervention in these rules allowing the programmer to coerce the type to another type before evaluation. Thus casts share the same level of precedence as grouping (brackets) operations. Casts in Tepal are indicated as in Java using the (<Type>) approach indicated in the example below:

```
aVal := (String)1 + "2"
```

Causing the number 1 to be cast to a String type, i.e. "1", before evaluation, thus resulting in "12" as the result.

Casting or transforming defines a conversion between one type representation and another, the result is destructive in that the object id of the result is different from the object id of the original, unless the type of the original is, or can be viewed as, the type of the cast, in which case there is no physical transformation performed.

An example of an operator declaration is shown below:

```
operators:
    +:
        //perform the operation
        result := [new, left.value + right.value] -> Self
    ;
    +=:
        //perform operation with respect to self
```

```
            value += right.value
        ;
        (+):
            pattern: left (+) self;
            precedence: 5;
            //do task
        ;
;
//System configuration of operator templates.
SystemOperators:
        +:
            pattern: left + right;
            precedence: 1 ;
        ;
        +=:
            pattern: self += right;
            precedence: 9 ;
        ;
;
```

Since Tepal effectively always converts to the same type for evaluation, all evaluations are of the same type. Thus once + is defined, there are no reasons to re-define it within the context of the Type. Note the inline definition of a new operator. This may not conflict with any other declaration of the same operator.

Conversion becomes very important. All conversions are either to or from a type. The example below shows how this is defined.

```
conversions:
        to Integer:
            result := self.value
        ;
        from Integer:
            self.value := right
        ;
;
```

From this it is possible to see the two keywords, i.e. *to* and *from*, the *to* implies a conversion from self to the value. Only destructive conversion need be defined, non-destructive conversions are implicate. When converting *from* Tepal will construct a new instance of the type and will evaluate the conversion in terms of the newly created instance. Tepal will perform a blank construction, thus it is important that any required value to described through a default will be populated in the process. Tepal also views the conversion as a left handed single parameter operation, as such right is available as the instance of the type to be converted.

Note that any pattern of symbols or characters can be configured as an operator. The only constraint on operators is that they may not clash with variable or Type names. An example of a character operator may be ‘instanceOf’, which would evaluate whether the left object is an instance of the right object.

Operators are scoped, that is an operator declared at system level is visible throughout the system, whereas an operator declared within the bounds of a Type is associated with the type. Thus it is possible to declare the same operator in different contexts with different properties, if a conflict is detected the execution environment will require run time type information to determine which instance of the operator is required to be evaluated (or vari-

able / operator resolution where that is obscured). This will obviously incur a larger penalty for execution, thus it is best not to allow such conflicts to exist where possible.

Events / Rules

There are two core mechanisms to initiate execution of code within the Tepal environment, the first is via the receipt of a message, the other is by triggering an event (or rule). Message passing is used to communicate between objects and is typically concerned with the performance of specific actions or sequencing events. Events are about monitoring the state of an object and, based on a change of state, performing a set of actions pertaining to that change of state. The types of events in the system are also specialised in that there are different types of events in the system, based on the type of event the configuration of the event rules and the nature of the actions associated to the events will differ. For example consider the following:

```
properties:
     name String required
     dateOfBirth Date required
     dateOfDeath Date
;
constraints:
     death:
          dateOfDeath == empty
       || dateOfDeath >= dateOfBirth
     ;
;
```

In the example above a constraint event is defined. A constraint event accepts no parameters, is limited to a single expression that must evaluate to either true or false and can only reference properties reachable via the current object.

The constrain parser is responsible for registering with the properties that are used as part of the constraint rule to receive change events. Thus when a property associated to a rule is changed then the rule is evaluated. In this case the action would also be supplied by the constraint implementation.

Another type of event is depicted below:

```
properties:
     value Number
     client Investor
;
rules:
     bigSpender:
          condition:
               value > 1000 && client.address in {'London', 'New York'}
          ;
          action:
               [new, client] -> BigSpender
          ;
     ;
;
```

In this example the condition identified the trigger for the event, and the action block identifies what should be done when the event is detected.

It is also possible to allocate events against messages as well, this is shown in the example below:

```
messageHandlers:
     new:
          expects:
               name String required
               dateOfBirth Date default: [today ] -> Date;
               dateOfDeath Date
               someProperty String
          ;
     ;
;
rules:
     onCreate:
          condition:
               message: new;
               someProperty == "Hello"
          ;
          action:
               result := [new, name:name]->MyType
          ;
     ;
;
```

In this example the condition includes the keyword message, this holds the message events that are to be used as input triggers. In this example when the new message is called on the surrounding type then the rest of the condition is evaluated. All the properties of the message are available, this includes the return property and the action is capable of changing the result returned. The example presented is not a very good example as it may not be very useful to call new on the same type again, better to just change the properties, but it does highlight the potential. If the messages variables are modified then this is assumed as being pre-message processing, the changing of the result value is considered as post-processing. The message handler is always called, given the dynamic nature of Tepal, rule based non-execution of an object should rather be handled by modifying the message handler. Generally, skipping the execution of a message is not considered good practice as there may be other dependencies to the messages execution that would no longer be triggered.

Rules can be directly associated to types, in this case the rules name space and scope are in terms of the surrounding type. Rules can also belong to rule collections. Rule collections allow for the collection of a set of related rules into a bundle. The rule bundle can define some default properties such as name-space and elements of context. In a rules bundle the individual rules can bind to any event type, this can cause objects to be extended dynamically adding, for example, new constraints to an object. An example of this is shown below:

```
rules MoneyLaundering:
     aliases:
          uk.finance.transaction.Transaction Transaction
          uk.finance.laundering.LaunderWorkFlow LaunderWF
     ;
     notifyOnLargeTransaction:
          message: Transaction.new;
          condition:
               value >= 10000
          ;
          action:
```

```
            //Note: self is with reference to Transaction
            [add, self] -> LaunderWF
        ;
    ;
    constraint Transaction.maxValue:
        value <= 1000000
    ;
;
```

In the example the rules bundle is captured in its own resource collection (i.e. File), and is given a name that uniquely references the bundle. The aliases section allows for setting up short names to commonly used fully qualified resources. Since the rule triggers with a message the rule is automatically scoped to the associated instance object. Thus all variables visible to the message are visible to the rule. This includes type references such as self. Rule bundles do not have self references as they are never instantiated, they can only be enabled or disabled. The constraint keyword allows for the dynamic addition of constraints to types, this requires the type on the left, separated by a dot, then the name of the constraint to add on the right. This will add the constraint to the type. All rule types can be added to a rules bundle, by default all elements are considered as rules unless explicitly advised otherwise.

Rules may need to be ordered in their execution, that is some rules may need to be processed before other rules, in order to support a notion of ordering the rule can be assigned a priority. The priority indicates ordering. The rule types have default priorities based on their nature, thus constraints have a higher execution priority than normal rules. Rules that mutate input values have a higher priority then constraints (since they will affect the result of a constraint).

It is also possible that a rule may be split into sub-rules, this occurs when the rule has multiple effects, such as changing input values as well as output values. In these scenarios the code is split to allow the execution of the different elements to be appropriately prioritised and executed at the appropriate points. This happens at compile time, in these cases it is not possible to override the default rule priorities. Any priority set will only influence the position of the rule with respect to the other rules at the different levels.

An example of this ability is depicted below:

```
rules:
    myRule:
        priority: high;
        ...
    ;
;
```

Possible values include: first, high, medium, low, last. Note that first and last should be used with care and first does not always guarantee being first. The indicators are treated as hints to the execution engine. It is also possible to formally ensure that one rule runs before another by explicitly stating the fact, e.g.:

```
rules:
    myRule:
        before: myOtherRule;
        after: thisRule, thatRule;
        ...
    ;
    myRule2:
```

```
        with: myRule;
        ...
    ;
;
```

In this example we see the use of 3 new keywords, namely the before, after, and with keywords. Before requires the rule to execute before the rule(s) declared. The after keyword indicates the rule should only trigger after the rule(s) declared. Finally the with keyword ensures that a rule will run either just before or just after the rule declared. If many rules associate to the same rule under with, then the rules will execute as if they each had exactly the same priority.

From an implementation perspective rules can be thought to be allocated to execution queues, where they are deposited until they can be processed. The execution list is a collection of execution queues, each queue is ordered by priority or specific before / after / with rules. The rules are then compacted before execution to ensure a rule is only triggered once given a change transaction.

Transactions

All property changes are controlled within the context of transactions. Transactions allows for atomicity of processing. There are two main types of transactions within the system, namely:

- **Event transactions -** Event transactions indicate a boundary points for event processing. This type of transaction is used to delay the processing of event matches until a set of operations are complete. This is used mainly as a performance optimization. But does indicate a sub-transaction boundary and is used for transaction checkpoint hinting.
- **Data transactions -** Data transactions consist of units of change logic that are grouped together in order to ensure that either all changes are successful or none of the changes are successful. Once a data change transaction completes successfully the sate of the object graph is ensured to be consistent and persisted (for those objects which require persistence).

Event transactions are nest-able in that a new transaction can be nested within an existing transaction, however, the event handling is always delayed until the last transaction completes. An example of the use of event transactions is shown below:

```
messages:
    myHandler:
        ...
        holdEvents
            ...
        releaseEvents
        ...
    ;
;
```

The holdEvents keyword indicates that a new event transaction should be initiated, the releaseEvents marks the end of the event transaction. This should be initiated when it is expected that a large number of changes are going to be made, for example when results from a screen session are being set on a number of objects. These boundaries should never encompass a long running transaction, i.e. this should not bound to large volume interactions or user interactions.

Data transactions are slightly more complicated as they can be long running, and need to support atomic data handling. The support must include the ability to start independent

transactions within transactions that commit separately as well as nested transactions. The system also requires to distinguish between read-only transactions that do not mutate the data-base. But do force a refresh of the data view. Examples of transaction handling is depicted below:

```
messages:
     myHandler readOnly:
          ...
     ;
     myHanlder2 forceNewTransaction:
          ...
     ;
;
```

By default all messages are bound within the current transaction, if their is no current transaction a new transaction is created. Marking a message as being readOnly ensures that the message will not be part of a write-able transaction. If the current transaction is a write-able transaction, a new read-only transaction is created, any changes made within the context of a read-only transaction are non recorded and are rolled back at the end of the transaction or if the transaction fails.

The forceNewTransaction ensures that the message will execute within the bounds of its own transaction. This is always a write transaction. All code execution from the point of execution of this message handler will be executed within the bounds of the new transaction.

Transactions are not limited to a particular thread, but rather belong to the execution of code and the events that are directly associated to the processing of the message. If properties are modified outside the context of a transaction, a new transaction context is automatically created at the level of the message handler that is co-ordinating the property change. If the property change is not been co-ordinated by a message handler, the change is committed directly (i.e. the transaction scope is determined to be the property change and associated events generated by the property change. This could become a performance problem, it is important that transaction boundaries are always correctly identified and managed.

//TODO: Is this condition ever achievable? Perhaps a better hieristic is possible?

Threading

Generally threading is controlled by the execution engine and Tepal programmers should not concern themselves with threading. From an execution engine perspective however, execution of message handlers and event handlers is performed by distributing the load of the execution across a number of threads / servers based on runtime configurations and a statistics engine that determines the execution cost of the various handlers. It is possible to provide hints to the engine as part of a deployment configuration. A deployment configuration is associated with an installation instance, every application should provide a basic configuration that can be overridden by the application administrator. The live configuration is stored separately from the code base and is can be configured from an instance and or server context.

The Tepal execution engine tracks objects that are active in different execution contexts (threads or servers) and will attempt to provide an appropriate locking level on objects to ensure that conflicting modifications of shared resources will fail as soon as a conflict is detected. Shared objects only exist within the context of a transaction and server combination, thus it is only these scenarios where resource locking is appropriate, if Tepal identifies

the potential for a transaction to concurrently modify a resource it will ensure the appropriate concurrency control is implemented (i.e. using a mutex or semaphor). Objects that are in different transactions are generally copies of the master object and as such do not need to implement active locking strategies. Shared objects living on different servers are always copies of the master object. In all these cases the approach taken to detecting access conflicts boil down to write-write conflict checking with fail fast semantics.

There are some side effects of threading / concurrency that should be dealt with by a programmer, these include:

- **Cost hinting -** The ability to programatically provide coarse hints to the system as to the potential cost of the method, the costs hints are limited to:
  - **inline -** This handler is low cost and is a candidate for serial execution within its execution context. By default all constraints are considered inline and all rules are considered local.
  - **local -** This handler is sufficiently complex to warrant execution within the context of a new thread, but not sufficiently complex to justify remote execution over a network.
  - **localServer -** This handler is sufficiently complext to warrant being called over a LAN or other fast network, but is not sufficiently costly to warrant WAN execution.
  - **server -** This handler is sufficiently complex to warrant server based execution of the handler, even in the presence of low bandwidth connectivity to the server.
- **Singletons -** Some types should be implemented as singletons and irrespective of cost should only have one running instance at any point in time, or at least have that appearance. This affects the approach taken to access the type instance as well as concurrency controls applied.
- **Concurrency exceptions** - The code may provide additional logic to handle concurrency exceptions. This can be provided in terms of declaring a standard approach to dealing with the exception, such as re-play (which will refresh the object set and attempt to re-play the changes to the system), or provide custom handlers that may attempt to identify the conflict and heal the system appropriately. The standard approach is to abort the transaction and propagate the error to the initial source of the transaction for it to handle or not handle.

Examples of the above include:

```
messages:
     //Mark the MyHandler as having cost of inline
     MyHandler inline:
          ...
     ;
;

//Mark the entity type MyEntity as a singleton instance
entity MyEntity singleton:
     ...
;

messages:
     MyHanlder:
          exception ConcurrencyException:
               strategy: replay;
          ;
          ...
     ;
```

```
;
```

Types

A type is the core building block of our instance objects. All instance objects have a type that they are patterned after. It is possible to modify the object after instantiation, however the initial instatiation of the object is always controlled through a type pattern. The type declares the initial structure of the object and its default properties and behaviour. In Tepal there is no single type keyword as the class keyword in Java or C++. Tepal types encode patterns of behaviour and allow the compiler to gain a better understanding of the intended use of the object being instantiated, for example:

```
entity Person:
    ...
;
```

declares an entity type object. The entity type defines an object that is capable of being persisted via an object reference. Most objects can be persisted, however, entity objects supports orthogonal persistence (that is it has persists as a single copy and all references are persisted as a reference and not as a copy). Objects that are not of type entity are persisted by value and not reference.

Different types also can restrict or add type body content as well as provide default behaviour not explicitly declared. For example an entity type automatically adds an oid property that refers to the persistent object reference used to uniquely refer to this object. This is different to the memory address of an object copy that uniquely refers to the specific copy in memory (the ??mid??).

//TODO: Example of a restricted type

Tepal provides a default set of types, it is possible for adding additional types to the language. The power of Tepal is manifest in the pattern based type mechanism. Each type supports its own collection of modifier used to configure the type, for example, the role type:

```
role Employee belongsTo Company appliesTo Person:
    ...
;
```

In this example the role type supports the belongsTo and appliesTo modifiers. Each modifier takes, as a parameter, an entity type. The belongsTo defines an ownership of the role as well as an indication of multiplicity (many-one style relationship). The appliesTo indicates the entity type that is to be collected, namely the Person entity type in this context. The appliesTo also indicates that by default any behaviour or properties defined will be attributed to the Person type as well as extending the Person type associated to this role with the Employee role, i.e. the Person will respond positively to the isa Employee question as well as support casting to the Employee type.

Role can also apply to an entity without the belongsTo as shown below:

```
role User appliesTo Person:
    ...
;
```

An example of where a one-to-one association is meaningful is:

```
role CEO associatedTo Company appliesTo Person
```

Note that the role association adds additional behaviour that is not normally captured in a normal property association, this is the notion of time-value properties, that is it should be possible to ask questions such as list historical Persons associated to the company in the role of CEO. This behaviour is not the same as the behaviour of a normal property.

It is also possible to enquire using the wasa key word to denote if an object has held the type in the past, i.e.

```
person wasa CEO
```

This is similar to the isa style keyword, that determines if the object currently holds the association.

The general rule to using a role is when you need to extend the behaviour of an associated entity with a characteristic that is typically time based or characteristic of an entity that is not a direct consequence of being the entity.

Properties

Properties like message handlers are fundamental to objects. Properties are values associated to objects that define their characteristic state. Properties usually are associated to a specific type. Examples of property definitions are presented below:

```
properties:
     name String[30] required;
     dateOfBirth Date required notModifiable;
     age Number derived:
          result := (dateOfBirth - Date.current).years
     ;
;
```

As can be seen a property is defined as the name of the property, then the type of the property (optional), followed by any modifiers. The modifiers declare the behaviour of the property. Examples presented show required, indicating that the property must be set on construction. The notModifiable indicates that the property, once set, cannot be changed. Finally the derived modified indicates that the property is derived, that is the property is calculated from existing properties.

Note the square brackets following the String object, this indicates the String object has a length constraint.

Primitive Types

There are none.

That is all values represented in Tepal are objects. There are however a number of standard transformations performed at the compilation phase of the source code. These include:

- **Numbers -** Numbers are recognized and parsed into constant Number objects
- **Strings -** Identified by the “ symbol bounding the string, converted to a constant String object.
- **Boolean values -** true and false

Examples of these are below:

```
myVar1 := “String 1” + “String 2”
myVar2 := 1 + 2.01E21
myVar3 := true
```

It is also possible to parse generic objects, for example:

```
myVar := (Date)”01 Jan 2004”
```

The mechanism indicated here shows a request to cast a String object to a Date object. The runtime environment will call the from String conversion method resulting in the required Date object.

Only a limited number of parser recognisable types can exist as the parser is required to be able to uniquely identify the type to associated the item to be parsed. In the example above it would be hard to uniquely identify the date representation as a date, it could be argued that the date should be represented using one of the other traditional formats such as dd/mm/yyyy, but this could also be interpreted as a number divided by a number divided by a number, since it is potentially ambiguous, we rather restrict the number of parse-able types and support advanced type recognition using the conversion mechanism. Since parsed items are constants it is possible for the compiler to optimise the compiled code to perform the conversion during the compilation process and thus avoid the additional overhead at runtime.