A new Generation Object Model and Language

Andrea Guerrieri
Via Franco Bolognese, 2
40128 Bologna (BO) Italy
(+39) 340 3969748
aguerrieri82@gmail.com

ABSTRACT

Object-oriented programming is nowadays one of the most familiar and widely used paradigms. As extensively argued in the literature, inheritance, a key feature for code reuse, fails in its aim. We impute this failure, to a basic conceptual mistake in the "is a" relation. After showing the weakness of some strategies to preserve expressiveness using single inheritance, we propose an alternative model based on services. This model helps to build an object-like system that is modular, extensible and reusable by means of essential, but powerful design structures. We also introduce the S language, through which we can easily express the service-oriented paradigm. Finally, we show how to emulate S features in an existing OO language.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features; F.3.2 [Theory of Computation]: Semantics of Programming Languages; D.1.5 [Programming Techniques]: Object-oriented Programming;

General Term

Languages, Theory, Design

Keywords

Inheritance, Service-Oriented, 'S, Components

1. INTRODUCTION

One of the most important aspects of OO, is the code reuse capabilities associated with inheritance. However, in many situations, inheritance alone is not enough. To overcome these limitations, several techniques has been proposed in the literature. Some, try to cover the extensibility problem, the ability to modify an existing class hierarchy, even without the original source code. Some other, the code reuse problem, the ability to compose a class library, combining elementary building blocks. On this side, are placed components models, such as CORBA[14] or COM[54], or new language-constructs, such as mixin[10] and traits[15]. However, almost all of these solutions, do not reject the concept of class inheritance, but instead, try to improve it. One more not-negligible aspect of software development is the portability, the ability to execute an application in different platforms. An increasingly popular solution are virtual machines / interpreters, sometimes accompanied with a wide class framework. This cannot be the answer to portability: system are different because provide different things, in different manners, and a software should be enabled to take advantage of the peculiarity of one system over the other. To obtain a more portable system, one way is to separate the "needs" (or services) from their realization. Using the interfaces and applying some design patterns, we are able to keep separated this two aspects. However, to properly describe a service-based system, we should use a language with a built-in service semantic. This language is called 'S, and it will be gradually introduced in this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGPLAN'05 June 12–15, 2005, Location, State, Country.
Copyright © 2004 ACM 1-59593-XXX-X/0X/000X…$5.00.

1.1 Contributions

In section 2 will be illustrated as the inheritance, and especially single inheritance, cannot represent the real world, generating code duplication. We will impute its failure, to a basic conceptual mistake of the "is a" relation, that does not take in account of the subjectivity and dynamicity. In section 3 will be shown some strategies built over single inheritance, to maximize code reuse. In section 4 will be introduced the service model, as an alternative to current object model. We will be explained how migrate from a class-hierarchical system to a class-based component system, on which the class concept goes into the background, to leave space to the more abstract and flexible idea of service. In section 5, will be presented an overview on 'S language, designed specifically to express the service paradigm. Finally, after a more accurate comparison with related works, we will show in section 7 a possible translation of 'S into an existing OO language, like c#. This paper does not introduce any formalization, however it can offer an intuitive idea, with a rich set of example and sample code.

2. CLASSES

Through classes, we can enclose in a single structure all objects having similar features: the presence of a particular feature cause that an object belonging to a specific class. The class establishes also the identity (or type) of the object, through the "is a" relation. If a subset of objects of a specific class has additional features (more specific), we can define a new class (subclass) inheriting all features of its superclass and including the new features.

2.1 Inheritance

Due to the well known and already discussed problems of multiple inheritance (first of all, the diamond problem), many of modern OO languages allows only single inheritance. Example 2.1 shows a simple case evidencing single inheritance failure:

Example 2.1 We have three class C1,C2,C3, and some common feature f1,f2,f3. We want to define a class hierarchy, thought which share duplicated features:

C1 = {f1, f2}; C2 = {f2, f3}; C3 = {f1, f3};

Moving f1 in a base class, would cause the duplication of f2 in C2, because C2 has no common base with C1.

B1 = {f1}; C1 = {B1, ***f2****}; C3 = {B1, f3}; C2 = {****f2****, f3}*

Moving f2 in a base class, would cause duplication of f1 in C3, and so on.

Conflicts like those arisen in Example 2.1, may occur quite frequently modeling large class systems, a concrete example is shown in Figure 1.

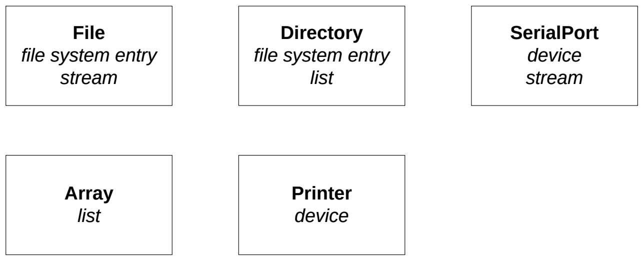

Figure 1. Some samples classes and their features. Some features are shared between unrelated classes, therefore is not possible creates a feature-based hierarchy without duplication.

2.2 Ambiguity on "is a" relation

Inheritance represent the "is a" relation, but for several reasons "is a" cannot model reality. What an object is cannot be established by a definition, because that is a subjective and dynamic concept. Subjective, because different points of view, different uses cases, may give to the same object, different identities. Dynamic, because in the future we might find a new way to use an existing object. Additionally, the identity of an object may also depend on its internal state. Regardless of the multiple identity that an object can assume in real world, each application affects only a limited portion of real, so an hypothetical Rect class in math applications, can be also different from a Rect class in graphical applications. Such limitation, anyway, does not resolve the "multiple identity" problem, as show in following cases:

2.2.1 Same object, different services

This occurs when an object exposes more services belonging to different domains.

Example A bicycle is a vehicle, but at the same time, can be used in a fitness club as an aerobic machine.

Example Physically, a serial port is an hardware component, so must extend Device class. But from a service point of view, a serial port is a Stream, that isn't a subclass of Device.

2.2.2 Same object, different views

This occurs when an object belonging to different classes in different domains, therefore may exist multiple views of same object. Similar analysis was advanced in [30]:

Example In e-commerce platform, a physical Printer may be represented by a generic Item class, exposing only attributes and methods useful for sale and delivery (price, brand, weight). In the device driver development, the same Printer can be represented by a Printer class, that may extend Device class, exposing only attributes and methods useful to print job (es. Print method)

2.2.3 Same object, multiple components:

This occurs when an object is an aggregate of independent components / features, therefore each component may be used as base class.

Example A mobile phone with embedded camera can be viewed as a specialized Phone (phone with camera) or as a specialized Camera (camera with phone). Anyway, is possible to reuse only one of two components.

2.2.4 Different state, different class

This occurs when the object class changes according its internal state.

Example Ellipse and circle are two different shape, belonging to different class. But when both radius of ellipse have same value, the ellipse turns into a circle.

3. LIMIT CODE DUPLICATION

Due to lack of single-inheritance systems, several technique and pattern may be used to limit code duplication. In this section are shown some examples, suitable mostly on static and strongly-typed languages (such as java, and c#). Will not be considered multiple inheritance, traits and mixin, which will be discussed later. If reader is already familiar with those techniques, can skip this section and move directly to section 4.

3.1 Strong base class

Some features are joined together in a common base class, and each different behavior controlled by attributes and flags.

Example There are different kind of buttons, some that differ from their aspect (es. textual, image, etc), some others that differ from their behavior (es. toggle, push). We can model the aspect with an attribute, and the behavior with the specialization:

```
public class Button {
   protected ButtonType _type;

   protected void Paint() {
      switch (_type) {
         case ButtonType.Image:
            ...
      }
   }
}

public class ToggleButton : Button { ... }

Public class PushButton : Button { ... }
```

As a design rule, a parameter should be used only if: (a) the underlying algorithm strictly depends to it or (b) parameter is not constant and its value cannot be determinate design time. If a function (or a class) have a different behavior, with a different parameter values and domain is limited, should be created a specific function (or a class) for each value of domain. For example, in following function, the format parameter defines the behavior:

```
string formatNumber(int value, string format) {
   if (format == “c”) return ...
   else if (format == “x”) return ...
}
```

therefore, would be better split `formatNumber` in two distinct functions:

```
string formatCurrency(int value) { ... }

string formatHex(int value) { ... }
```

If the class capabilities depended on its internal state, the type concept itself would fail, because what make two type different is exactly what they can do. Even if sometimes, some operations are not allowed with some states, the state cannot change the nature the object. Besides the conceptual problem, there are several disadvantage using this approach:

Extensibility: All different behaviors, are hard-coded into a single class, adding a new variant requires to have access to source code.

Memory: Unnecessary memory allocation to maintaining on state something that will remain constant for all object lifetime.

Maintenance: The large concentration of different behaviors on a single structure, make the source code harder to understand and maintain.

Performance: Code runs slowly due to presence of numerous flow control instructions, required to perform different action with different state.

The C++ templates[18] with constant parameters, avoid both memory and performance problems, because each instance of a template with different parameters, generates different code (and types), and in optimization stage all constant variables / expressions / statements can be removed . For example:

```
#define IMAGE 0

template <int TYPE>
class Button {
   protected: void Paint() {
      switch(TYPE) { ... }
   }
}

Button<IMAGE> button;
```

Conversely, use of templates requires the distribution of source code.

3.2 Composition and aggregation

Some features shared across unrelated objects, are implemented in separated classes. A class needing such features, can include a reference to the component-class in a private field, rather than inherit it. The host class can choose to expose directly the component-class, or create wrappers only for specifics methods. The situation described in Example 2.2, can be modeled as follow:

Example SerialPort class can extend Device class, and stream service can be implemented in a separated class (SerialPortStream) extending Stream. SerialPort can include an instance of SerialPortStream in a private field.

```
public class Stream { ... }
public class Device { ... }

public class SerialPortStream : Stream {
   public SerialPortStream(SerialPort sp) { ... }
}

public class SerialPort : Device {
   protected SerialPortStream _stream;

   public SerialPort() {
      _stream = new SerialPortStream(this);
   }

   public SerialPortStream Stream {
      get { return _stream; } }
   }
}

SerialPort port = new SerialPort();
port.Stream.Read(...);
```

It was possible to express a situation that single inheritance couldn’t express without code duplication, but with following disadvantage:

Accessibility: Generally the components need to access to private member of host class, and vice-versa. Therefore, many languages allow to invade private space of classes (es. friend keyword in c++, internal keyword in c#), breaking the block box concept on which object-programming should be founded.

Memory: Each component is allocated in a separate memory block, increasing the heap fragmentation.

Typeing: A component class is not a subtype of host class. This makes composition weaker than inheritance.

3.2.1 Transparent composition

A class do not expose directly its components. All methods provided by components that must be public, are re-declared in host class. Host class methods, redirects the call to associated component method, giving back the return value to caller (*aca* delegation pattern [2])

```
public class SerialPort : Device {
   protected SerialPortStream _stream;

   public SerialPort() {
    _stream = new SerialPortStream(this);
   }

   public void Read() {
      _stream.Read();
   }
}
```

This allow to hide that some features are implemented by external components, but subtyping problem still persist.

3.2.2 Transparent composition with interface

Classes and inheritance are used primarily as code reuse blocks, almost all of the typing tasks are delegated to interfaces.

Introducing IStream interface, both Stream and SerialPort can implement IStream:

```
public interface IStream {
  void Read()
}

public class Stream : IStream { ... }
public class SerialPort : Device, IStream { ... }
```

3.3 Helper class

Each class provides a minimal set of features, any additional features are implemented by helper classes. An instance of helper class must be created each time is required, usually passing the reference of target class on constructor.

For example, The Stream class may expose only basic functions (es. read a bytes block from current position), and delegate to an helper classes all advanced tasks::

```
public class Stream {
   public void Read() { ... }
}

public class StreamReader {
   protected Stream _stream;

   public StreamReader(Stream stream) {
      _stream = stream;
   }

   public string ReadLine() { ... }
}

StreamReader reader;
reader = new StreamReader(new Stream());
reader.ReadLine();
```

Since there is a different helper class instance for each target class, helper class can have own state.

3.3.1 Static methods

To avoid helper class creation, helper methods can be static. For example:

```
public static class StreamReader {
   public static string ReadLine(Stream stream) {
      do_something_with_stream
   }
}

Stream stream = new Stream();
StreamReader.ReadLine(stream);
```

This allows to save memory and to have a quicker access to helper functions, but static methods are in-fact equivalents to functions inside a namespace. This can affect encapsulation principle, since a behavior specific for a class is outside the class boundaries itself. Additionally, there are following limitation:

Overriding: Static methods doesn’t allow override: since the helper classes can be also used by any subclass of target class, could exist a specialized version specific for that subclass.[1]

[1] We can use method overloading to differentiate function based on type. Unless we does not use a dynamic dispatch[13] it would fail, for example, when a reference of type A holds an object that is subtype of A

State: Can be expensive keep state between different calls on a multithread environment, if source language doesn't support local thread storage.

3.3.2 Conclusion

Helper classes can be used to add new class-features when source code is not available, or to share functionality across different classes. In cases where it can be applied, the use of the composition is preferable. If more than one helper class works at the same time in the same target class unknown interactions between the helper and target class may cause side effects.

Example In .net framework, text reading is separated from binary reading, because text reading can be performed also without streams (es, reading from a string). In such situation, if we have a stream with hybrid content (es HTTP), we must use different readers for different content type. If text reader use an internal buffer, any read operation may get more data than they have been requested, exceeding the text content boundaries. If stream does not allow to seek, when we switch to binary reader, some contents could be lost.

---

4. SERVICE MODEL

The idea to build a class system, using elementary unit, that can also works cross-cut to class hierarchy, is popular in literature [44][30][53][28]. Even with different names, all concern, features, and aspects, perform a kind of decomposition, in order to (a) extend and control an existing class system (b) to share the behaviors among unrelated classes. We want to introduce simply another way to obtain in, using the more intuitive concept of *service*. Of course, the concept of service is not new. For instance, such approach has been successful applied to distributed systems, through the web services (*aca* SOA [32]). In this circumstance, services has been used as a strategy to interoperate between different platform. Even some component oriented model uses a service-like approach, one of that is COM[54].The most interesting principle that we can found on COM, is the separation of interface from implementation: a program ask and depend on abstract interface, without cares about the concrete class in which is implemented. More generally, this is known as *The Dependency Inversion Principle* [38]. Implement a system based on this principle, may requires several infrastructural code, to obtain an high degree of flexibility, and avoid code duplication. That is, because many interface would be implemented in a very similar way, and unless we use a multiple-inheritance-like (virtual inheritance, traits, mixin, etc) language, a class can reuse only the code pertains to one aspect. The key, is to leave behind the old class-hierarchical approach, to adopt a more effective service-oriented object model.

4.1 Introduction

We will introduce services, trying to give a definition of a well-know real-world object: *what is a table*? Is that piece of furniture? A car bonnet is a table? We can say that a table is an object that can be used as a table, that is an object able to provide a large and flat surface. This may suggest that the identity of an object depends more to how that object is used, rather than from its physical structure. In real life, we are all able to readapt a procedure, also using objects other than the original ones.

Example If we want to draw a straight line with a pencil, and we do not have a ruler, we may use any object with a straight and raised edge. The ruler is just one of many possible objects that can provide that service, and not the service itself. In different context, ruler may be simply a measuring tool, or a piece of plastic. Additionally, you can call 'ruler' any object that can assume that role at that time.

That is because we humans think "services", associating to a concrete object a particular behavior, and founding on that behavior our processes. A frequent misunderstanding that occurs in object-oriented model, is to exchange the service-provider (class) for the service itself, therefore programs, rather than depend on abstract "needs", depends on their implementation.

Example An application needs to write some logs. Initially we choose to use directly an instance of FileStream, and write logs calling its methods. Later we create a specific Logger class to manage file open / close stuffs, exposing a transparent log functionality.

Logger class is not the log service, is an entity (any of possible) <u>providing</u> a log service. This distinction may appear trivial to reader, but will mark the difference between our model and existing object model. In summary, any programming model that aim to represent objects, must take in account following principles::

An object is an aggregate of elements, each of which confers to it functions and attributes. Same elements may be found even in different and unrelated objects. (b) An object may provide one or more services, different objects may provide same service, and same service may be provided by different objects. (c) An object is often used for what it does, and not for what it is. A process depends on the service that a particular object can provide, and not from the object itself. (d) Is always possible find a new way to use an object, and an object can be used even differently than its original purpose.

4.2 Features

We shown in 2.2, how the same object can assume different identities. From now on, we will call each of that potential identities, a *feature* of that object. More general, we can define a feature as a capability of an object, a service that it can provide, a behavior, that responds to the question “what the object can do?”.

Definition 3.1 Feature *f2* depends on a feature *f1*, if *f2* cannot exist without *f1*. If feature *f2* depends on feature f1 in one object, then *f2* must depend on *f1* in all object that include *f2*

Definition 3.2 Feature *f1* is independent from feature *f2*, if exist at least on object that have *f1* but not *f2*

Definition 3.3 An object with at least two features mutual independent is called aggregate.

Rule 3.1 If two features are mutual dependant, then must be grouped in a single feature.

Rule 3.2 If exist at least one object that use only a subset of a feature, that part must be isolated in a distinct feature.

4.3 Migrating to services

Independently of the presence of different features, classes expose a *flat view* of objects, hiding their composite structure. In this section will explain how we can migrate from a hierarchical class model to a service-based model. The process consists of four steps: (1) features detection (2) class decomposition (3) services extraction (4) class composition

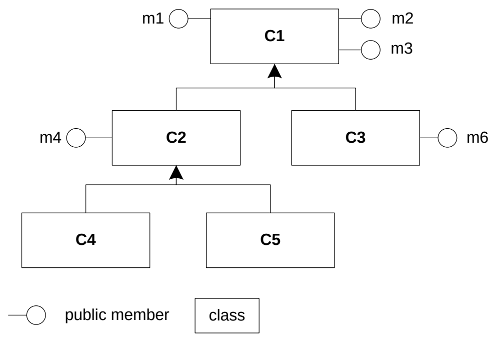

Figure 2. Sample class hierarchy

In next example, we will use following formalism:

$A \preccurlyeq B$	*A* is subclass of *B*
$A = \{F_1[M_1, .., M_n], .., F_n\}$	*A* is class, *F* is a feature of class *A*, *M* is a public member belonging to *F*.
$A \rightarrow B$	Feature *A* depends on feature *B*
$(F_1, .., F_n) \Rightarrow S(L_1, .., L_n)$	Feature F_n on class-space is translated into feature *S* on feature-space, and labeled as L_n
$A \multimap B$	*A* is based on *B*
$L@S$	Feature *S* with label *L*, on feature-space. Or service S implemented by L in service-space.
$S[A_1 \multimap B_1, .., A_n \multimap B_n]$	Feature *S* with label A_n is based on Feature *S* with label B_n.

Features detection Each class is decomposed in its primary *features*. Initially, we can simply associate to each class a distinct feature. Figure 2 shows this class hierarchy:

$$C2 \preccurlyeq C1;\ C2 \preccurlyeq C1;\ C4 \preccurlyeq C2;\ C5 \preccurlyeq C2$$

That can be initially decomposed in:

$$C1 = \{F1[m1, m2, m3]\}; C2 = \{F2[m4]\};$$
$$C3 = \{F3[m6]\};\ C4 = \{F4\}; C5 = \{F5\}$$

Applying rule 3.2, some features may be split in more parts. Supposing that *C2* depended only on m1, *C3* depended only on m2 and m3, and m1 used some private members of *C1*, we are facing to three different features. Renaming:

$$C1 = \{F1[m1], F2, F3[m2, m3]\}; C2 = \{F5[m4]\};$$
$$C3 = \{F4[m6]\};\ C4 = \{F6\}; C5 = \{F7\}$$

After we have detect all features, we must find their dependences. if a feature *A*, uses something of feature *B*, than *A* depends on *B.* In our case:

$$F1 \rightarrow F2; F5 \rightarrow F1; F6 \rightarrow F5; F7 \rightarrow F5; F4 \rightarrow F3$$

Figure 3 shows the final result.

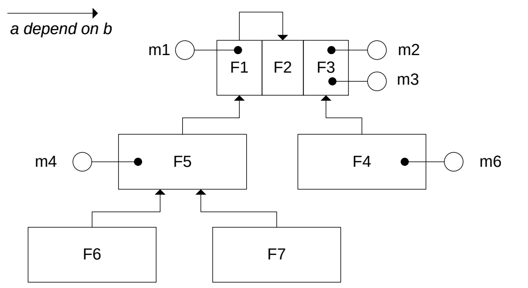

Figure 3. Result of feature detection.

Class decomposition In this stage, classes are temporary decomposed, every feature leaves the classes space, starting an independent life. We are now in the *feature space*, a kind of limbo between object-oriented-model and service-oriented model. All features without any public members, are grouped in a single namespace, and any relation of dependence between features of the same namespace, changed in “is based on”. That because, if a feature does not adds nothing of new, means that simply is changing the feature on which depends. In our example, only *F6* and *F7* are empty, and both depend on *F5*, therefore *F5*, *F6*, *F7* must be grouped together:

$$F1 \Rightarrow S2; F2 \Rightarrow S1; F3 \Rightarrow S4; F4 \Rightarrow S5;$$
$$(F5, F6, F7) \Rightarrow S3(A, B, C)$$

And changing also all relations:

$$S2 \rightarrow S1; S3 \rightarrow S2; S5 \rightarrow S4; S3[B \multimap A, C \multimap A]$$

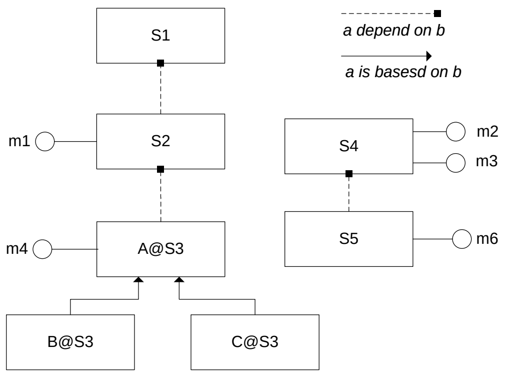

Figure 4. Result of the class decomposition

Services extraction: this is the most important conceptual step: all founded features with unique namespace, will form the abstract services (only as interface definition). Each labeled feature inside the same namespace, will be a particular implementation of that service. Every dependence relation must be reinterpreted. Generally, all dependencies between features extracted from two related classes, generate an interface (or conceptual) dependence. Instead, all dependencies between features extracted from the same class may generate an implementation (or concrete) dependence[2]. In our example *S1*, *S2*, *S3*, *S4*, *S4*, will be the services. *A*, *B*, *C* three different implementation of *S3* service. The feature *S1*, *S2*, since are originated from the same class, will be in a concrete dependence, all others in a conceptual one. Final result is shown in Figure 5

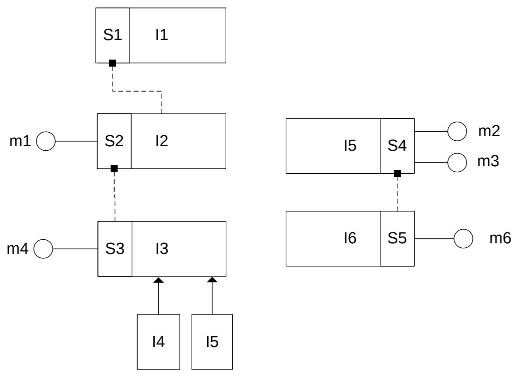

Figure 5. Result of service extraction

Class composition: All service implementations extracted from previous step, can be used as a buildings blocks to compose the new classes. The composition is valid only if all dependences are satisfied. For example:

$$C4 = \{I2@S2[m1], I4@S3[m4], I5@S4[m2, m3]\}$$

Class C4 is functionally identical to the old one, as is shown Figure 6

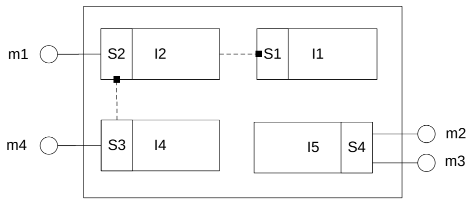

Figure 6. Equivalent of class C4 in service-model

In the service-oriented model, say that *"service B depends on the service A"*, can mean:

(a) If You can do (a generic) B, you can also do (a generic) A
(b) To do B (in that way), I need to use (a generic) A.

(a) establishes a conceptual dependence between services, something that is always true, regardless of the implementation. (b) establishes a concrete dependence, something that is true only for a particular implementation.

Example *"A list is enumerable"*, this is conceptual and always true. We can say that *"the list service depends on Enumerator service"*, or to be more colloquial that *"if you act like a list, you must also act as an enumerator"*. *"A list is sortable"*, is not always true, some lists can be read only, or contains element on which sorting cannot be applied.

The point (b) can be also read as:

(b*) If you can do (a generic) A, you can also do B (in that way)

We will define (b*) as "inversion of dependence", the ability of an object providing a service, to "inherit" all services-implementation that depend on it.

In object-oriented model, inheritance does not take in account this important distinction, with the result that each conceptual dependence is reflected also in a concrete dependence. Say that *"B is a specialization of A"*, does not necessarily imply that both must implements shared behavior in the same way. In fact, using virtual functions and polymorphism, B can alter the behavior of A, but this approach can generate code duplication[3].

[2] This distinctions will be clarified in next paragraphs.

[3] A changes made overriding a superclass method, may be suitable also for classes belonging to a different inheritance axis

In conclusion, what today was *"class B inherit from class A"* becomes *"service B depends on (require) service A"* or *"service B can be provided using A"*

4.4 Service based design

To design a system in a service-based environment, is not necessary to apply the procedure described in 4.3. We can easy decompose our application domain answering to this three questions:

1. What I need to do?
2. In which way I can obtain it?
3. Which entity can provide such service in that way?

Answer (1) produce *services*, (2) *service dependence* and *implementation*, (3) *classes*.

A conceptual service is represented by an interface, so we can use the term *interface* in place of that one of *service*. For each conceptual service, can exists many possible control interface, and many way to implement each interface.

Example a stream is an abstract service through which read and write data. Interface IStream with methods Read and Write is a way to control such service. FileStream is an object that provide (implement) IStream service.

In object-oriented model, interfaces can be implemented only inside a class, and a class can implement one or more interfaces. Since the unit of reuse is the class, is not possible to share an interface implementation between classes belonging to different inheritance axis.

In the service-oriented model, a service implementations, acts as building blocks, and the classes as a place (not the only one) in which aggregating such blocks, so that same implementations may be shared among different classes. As constraint, a service implementation can be used inside a class, only if target class provides also all the dependent services[4] .

A key feature of this model is that classes cannot inherit, but simply acts as container on which services can be realized and compounds.

4.5 Composition samples

Figure 7 shows some composition schema, taking advantage of service composition:

Schema I: (double implementation) B implement interface I1. A implement I2. Class expose both I1 and I2.

Schema II: (single local dependence) B implement interface I1 and require interface I2. A implement I2, so can satisfy B requirement. Class expose both I1 and I2.

Schema III: (private implementation) Similar to Schema II, except that class doesn't expose interface I2, that is used only to satisfy B requirement.

Schema IV: (external dependence) Class C2 implement I1 by B, but B require interface I2. Class C1 implement I2 by A. Class C1 is parent of C2, so can satisfy B requirement.

Schema V: (mutual local dependence) A implement I2 and require I1. B implement I1 and require I2. Both requirement can be satisfied each other. Class expose both I1 and I2.

Schema VI: (shared requirement) C implement I3 and require I2. B implement I1 and also require I2. A implement I2, so can satisfy B and C requirement. Class expose only I1 and I3.

Schema VII: (double local dependence) C implement I1. A implement I2. B implement I3 and require both I2 and I1. C and A can satisfy B requirement. Class expose all I1, I2 and I3.

Schema VIII: (interface dependence) Interface I2 depends on I1. A can implement only I2, creating an implicit requirement on I1. A expose both I1 and I2, but I1 is implemented through B provided by class C

[4] This is not properly true, a more precise definition is given in section 5

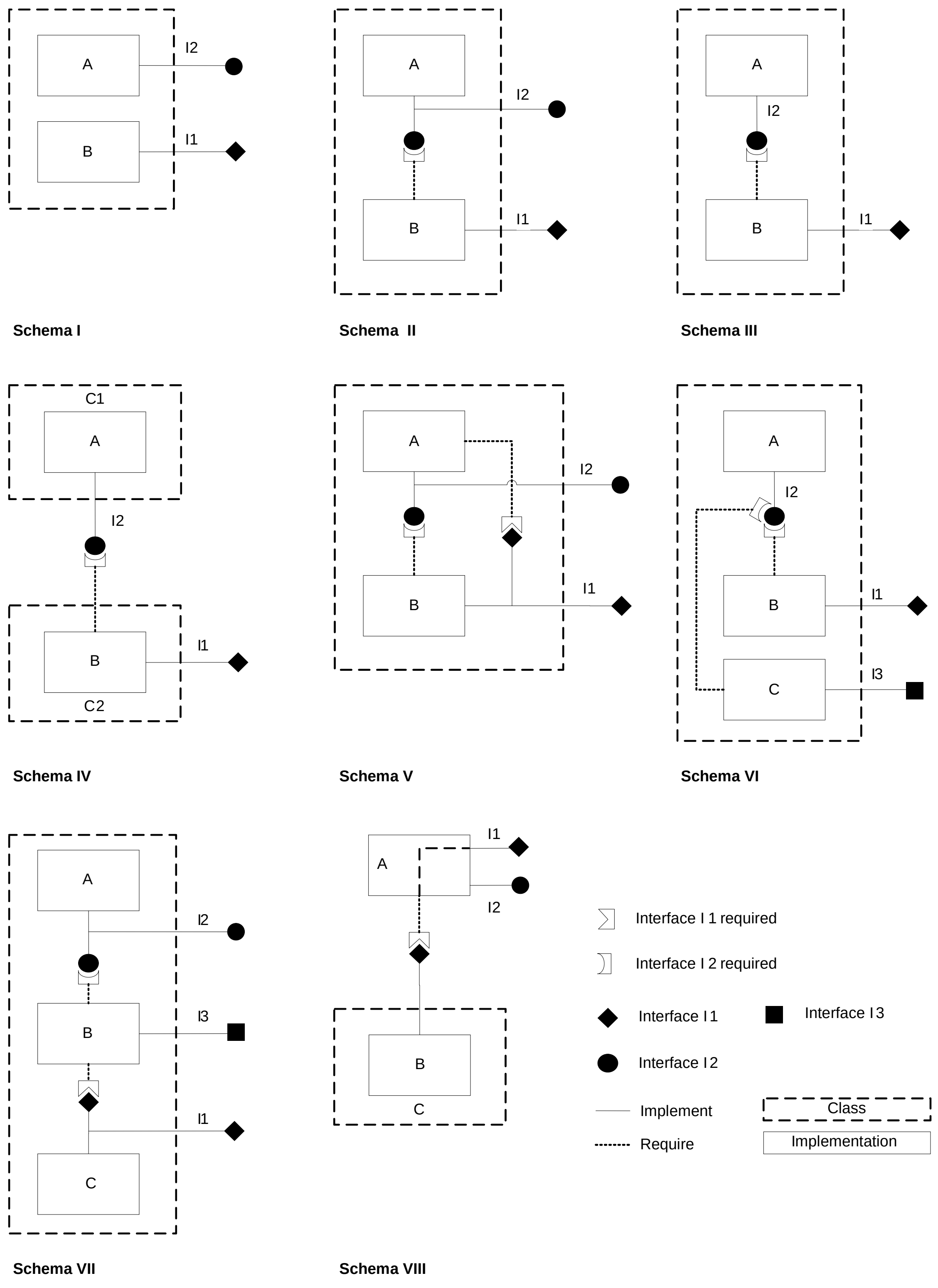

Figure 7. Examples of service composition

5. "S" LANGAUAGE

This section is only an overview of 'S language, showing mainly its aims and purpose, without giving a formal or rigorous definition. 'S will be introduced in this paper, to give to the reader a reference language to better understand the service model potential. The first goal of 'S design, is the ability to compile a code that is fast, and with a low-memory-usage, in order to be executed even in a embedded system or a microcontroller. Some theoretical features could be removed due their cost, but any decision can be taken only after the analysis of the profiling results on real world applications. Many features, statements and constructs are inspired from existing languages (such as c++, java, c#, and so on), therefore will be used without giving any definition: only the ones that diverge from status-quo will be explained and defined.

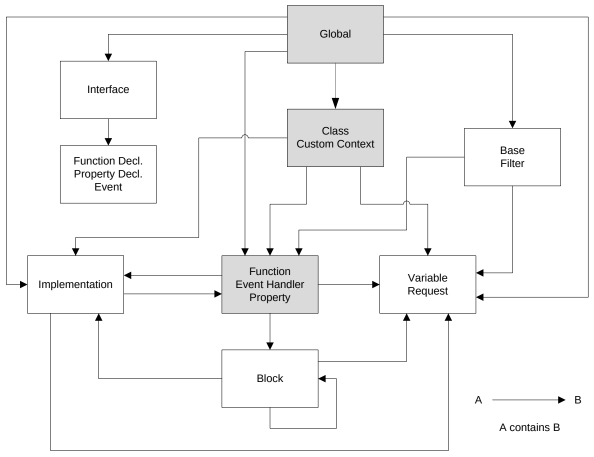

Figure 8. Overview off principal language construct, and their relations. The gray blocks defines also a context.

5.1 Hello world

An overview of the main 'S constructs and their relations, is shown in Figure 8. Nevertheless, we want to start introducing 'S, using the evergreen "hello world" sample:

```
require IModule

on IModule.load()
{
    require IConsole as c;
    c.WriteLine("Hello world");
}
```

By `require` statement, we are asking to the system for a particulat service. All requests declared in global scope, are resolved on program startup, and generally are directed to those services consumed in several parts of the program. `IModule` service is implemented by default from compiler, and rapresent the program[5]. Subsequently, it was written an event handler for `Load` event of `IModule` service. This event will be raised by system, and can be considered in fact as the entry point of the program. Inside the event handler, was required the console service (`IConsole`), and using the `as` clause, was specified an alias through which get access to this service. Finally was invoked the `WriteLine` method of the console service, in order to emit the string "hello world"

5.2 Type

Similar to c#, there are two main categories of types: value types and reference types. A schema of all types categories is shown in figure Figure 9

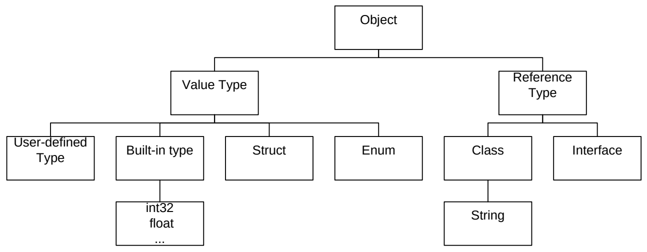

Figure 9. Summary of types defined in 'S

Value types are always passed by value (deep copy), and are usually stack allocated. To this category appertains Primitive Types (int, bool, float, etc), struct, enum and user-defined types. The only exception is for the string type, that is considered a built-in type, but is passed by reference.

Reference types, are always passed by reference (Shallow copy), and are heap allocated. To this category appertains classes and interfaces. Both the 'reference passing' and the 'heap allocation' they allow to a class type to be more extensible than a value type, because its memory size may be unknown at compilation time.[6]

5.3 Variable and Function

We can declare a variable in any point of program, using keyword `var`. If a variable is initialized on declaration, the type can be omitted (`a`), since it can be inferred from the assignment expression.

```
var x : int32;
var y : int32 = 10;
var z = 10;                                          (a)
var z1;      //illegal
z = "text";  //illegal, z is an int32.
```

Also a function can be declared in any point of program, even inside a function itself. In this release, a function does

[5] By program we mean this particular module, because a program can be composed using different modules.

[6] This advantage make sense only using certain compilers

not define a type. A function can require input arguments, and optionally can return a value. A function can be overloaded, so can exists many function with same name, but different arguments[7]. An argument can be optional, and in this case a default (constant) value must provided. All arguments that follow an optional argument, must be optional as well. A function must be called using parenthesis, and passing all not-optional arguments, separated from comma, in the same order on which they appear into the function.

```
function ToHex(value : int32,
               digitCount : int32 = 8) : string
{ ... }
function Reboot() { ... }
var result1 = ToHex(10);
var result2 = ToHex(10, 4);
```

5.4 Interface

The central point of any program written in 'S, are the interfaces. An interface describes a service, and a class is composed by service implementations.

```
public interface IStream {
   function Close():
}
public interface IInputStream : IStream {      (a)
   property IsEof : bool;
   function Read(buffer : byte[],
                 count : int32) : int32;
}
var x : IInputStream;
x. Close();                                    (b)
```

An interface can declare many prototypes of functions, properties, and events. An interface can depend on more interfaces *(a)*. That means that the class[8] in which interface is implemented, must also implement all dependent interfaces. Due to this constraint, is possible to have a direct access to all members of dependent interfaces, as if they were its own *(b)*. if interface A depend B, and B depend on C, then A depend on C. To create a conceptual dependence for an already defined interface, is possible to use `assert` statement:

```
assert IObject : IFormattable;
```

The module that declare assertion, must provide an implementation for all type that directly, or indirectly implement involved interface.

[7] With conditional function, we may have more than one function with the same prototype, this we will explained later.

[8] The class is not the only one structure by witch implement an interface

5.5 Base implementation

Is possible to share a full or s partial interface implementation, using the `base`. A base is not a type, so cannot be directly instantiated or referred.

```
public base BaseStream : IStream {
   var _closed : bool;                          (a)
   final function Support() {... }              (b)
   public function Close() {
     if (!_closed)
       _closed = true;
   }
}
public base EnahncedStream : BaseStream {     (c)
   public override function Close () {        (d)
      Support();                              (e)
      base.Close();
   }
}
```

A base can inherit from an existing base *(c)*, and in this case, all the well-known features of single inheritance are available[9]. Is possible to declare any support function or state *(a)(b)*, however, those members will be accessible only from the base, or from a base that inherits from it *(e)*. All the functions / properties can be overridden by default. In order to override a function, the `override` modifier must be specified *(d)*. To disable this feature, the base class can declare the member as `final` (b). If a support function is declared as private, then it will be not accessible to its superclass[10]. All interface members must be publics. Similar to the `implement` statement, also the `base` statement can implement only one interface. If this interface has some dependences, it will be possible call and use such member as if they were its own. The compiler will ensure that all dependences will be always resolved.

5.6 Class

The first place on which an interface can be implemented, is a class. A class can implement multiple interfaces, and for each interface, can select the base from which starting from. The class must implement all member of each interface that have not yet been implemented, and must also implement all dependent interfaces.

```
public class File {
   public construct(name : string) { ... }     (f)
   public implement IInputStream
                use BaseInputStram;            (a)

   public implement IStream{
      public function Close () { ... }
   }
   public function Close() { ... }             (b)
}
```

[9] Inspired to that one of c#

[10] Will be used the terms superclass / subclass improperly referred to the base implementation hierarchy

```
var x : = new File("foo.txt");                    (g)
x.Close();                                        (c)
x.IInputStream.Close();                           (d)
(x as IInputStream).Close();                      (e)
```

All properties not explicitly implemented, will be auto-implemented by compiler, by declaring a private variable on which store / retrieve the state. If `implement` does not declare any member, the block {} can be omitted *(a)*. Is possible to get access to an interface member in two way: *implicitly (c)* as if it were a direct class member of the class; *explicitly (d)*, by specifying the interface name. A public function that is declared into the class, has the precedence on any interface members *(b)*. In this case, or in the case of ambiguous member access (similar prototype in two distinct interface), is required an *explicit* access to resolve the ambiguity *(d)*. Implementing an interface as `protected`, all members will be protected too, except those explicitly declared as public. Same semantics for `public` (all public except those declared as protected). A protected member is accessible only explicitly. If all member of an interface are protected, an explicit access to this interface will be no more available. However, is still possible execute a cast to that interface *(e)*. An interface member cannot be `private`, however, whole interface implementation could be. A private interface will be never accessible, even with a cast. If an interface is implemented as private, also all interfaces that depend on it must be private. A class can declare optionally a constructor using `constructor` keyword (f). Is possible to create a class instance, using `new` operator (g).

5.7 Context

The class is not the one place on which implement a services. By `implement` statement, is possible to implement a service in different places, called *contexts*. A context can be defined as a place in which an interface can be implemented and required. The class, is only a particular kind of context, that can be "replicated" in multiple copies.

```
//global context
public implement ITextWriter use Console;       (a)
public class Console {
  //class context
   public implement ITextWriter { ... }          (e)
}
public function Foo() {
   //function context
   public implement ITextWriter { ... }
   Foo2();                                      (c)
}
public function Foo2 () {
   require ITextWriter;                         (d)
}
on IModule.load() {
   require ITextWriter;                         (f)
   Foo();                                       (b)
}
```

Above a summary of all defined contexts, that are *global* (the space outside any defined structure), *function*, and *class*.

Is very important understand the meaning of implement statement. When is declared in *global* or *function* context, is not simply a definition. More than define an interface implementation, it define a rule, asserting that that service must be implemented in that way. *(a)* must be read as *"if someone requires `ITextWriter`, uses `Console` class to provide it"*. The contexts form a stack, which has *global* as its base. Every function call cause the related context to be pushed into the stack, and then popped back when function returns.

```
public context ConsoleContext {                  (a)
   public implement ITextWriter use Console;    (b)
}
on IModule.load() {
   //global context
   enter ConsoleContext {                        (d)
      //UDC context
      require ITextWriter;                       (c)
      ITextWriter.WriteLine("hello");
   }
   //global context
}
```

By user defined context *(a)* (UDC), is possible to create a set of configurable environments, on which declare the default implementation *(b)* for a group of services. UDC have a similar role of a singleton or static class, but contrariwise, its content is not directly accessible. Prior to use any service or variable declared into an UDC, you must enter in such context. By `enter` *(d)* keyword is possible to declare a code block that runs inside a specific context. After entering, the old context is pushed into the stack, and the specified UDC becomes the current one. That means that any service request *(c)*, until we are inside, will pass through the UDC. There are no limit in nested `enter` call, even if they affect the same context several times.

5.8 Require

By `require` keyword is possible to ask for an interface, in any point of program. The required interface will be accessible as any normal variable, with the same identifier of the interface. Is possible to define an alias, using the `as` keyword. A request is resolved once `require` instruction enters in the scope, and not in the position in which is declared. All requirements declared in the global context, will be resolved once the module and all its dependences has been loaded. All requirements declared in a class or implementation context, will be resolved on construction. All requirements declared in a function context, will be resolved at each call. Due to is state-like purpose, is not possible declare a requirement inside a property. Normally, the request lookup begins by the requiring context, and then passes down through the context stack, until it reaches that one that can satisfy it. Is possible to alter this behavior, by specifying the reference context. Possible values are `local` (only local context), `global` (only global context), `ancestor` (any parent context but not local), or an identifier of any user define context.

```
public class Console { ... }
public class Debugger { ... }

public context ConsoleContext {
   public implement ITextWriter use Console;  (i2)
}

public class File {
   public implement ITextWriter { ... }       (i1)
   public constructor File() {
      require ITextWriter;                    (f1)
      require local ITextWriter;              (f2)
      require ancestor ITextWriter;           (f3)
      require global ITextWriter;             (f4)
      ...                                     (b)
   }
}

public implement ITextWriter use Debugger;    (i3)

on IModule.load() {
   require ITextWriter;                       (g1)
   require Console ITextWriter;               (g2)

   enter ConsoleContext {
      require ITextWriter;                    (c1)
      require ancestor ITextWriter;           (c2)
      require global ITextWriter;             (c3)
      var file = new File();                   (a)
   }
}
```

The runtime context stack in *(b)*, after the constructor call in *(a)* is `global→Console→File`. *(f1)(f2)* are statically resolved in *(i1)*. *(f3)* is resolved at runtime in *(i2)* (the first ancestor implementing `ITextWriter` is `Console` context). *(g1)(c3)(f4) are* statically resolved in *(i3)*. Even *(c2)* is statically resolved in *(i3)* (its parent is `global`, and does not depend on call stack). *(g2)* is resolved statically in *(i2)*

Any requesting interface marked as `local`, it will be also accessible in the calling context using `this` accessor. This make sense, because local constraint ensure that the implementation will be performed in the same context in which the request was made. Even in this case, any ambiguity can be resolved by specifying the interface name.

```
require local IConsole;
on IModule.load() {
    WriteLine();                              (a)
    this.WriteLine();                         (b)
    IConsole.WriteLine();                     (c)
}
```

(a) due to the `local` modifier, `WriteLine` of `IConsole` is mapped in local context (`this` can be omitted). The compiler will ensure that all requests will be resolved at runtime, by performing a static check at compilation time. That means that must exist at least one context, statically ancestor of the requesting context, that is implementing requested interface. The search is extended also at global context of all dependent modules. Is possible to ask for a new instance of the service (not shared with anyone else), using a `new` keyword. For the same principle, is also possible declare a new instance of an object, specifying only the required interface. Even when we are implementing an interface, is possible to specify as use clause, the context in which get the implementation by which starting.

```
require new IConsole;
var x : new IConsole;
public class TextBox {
   public implement IWidget use new global {   (a)
   }
}
```

(a) Means that the base implementation of `IWidget` must be retrieved from global context, that must be return a new instance;

5.9 Extension

```
Public extend class File {                     (a)
   public override implement IStream           (b)
                        use base { ... }       (c)
   public override implement ITextReader
                        use BaseTextReader {  (d)
   }
   public implement IItem {... }
}
public extend type String {                    (e)
   public implement IComparable { ... }
}
```

By `extend` keyword *(a)* is possible to alter a type defined into a compiled module on which we are dependent. `Extend` can be applied to a class, primitive type and UDC. To override an existing implementation, the `override` modifier *(b)* must be specified. When we are overriding an existing implementation, we can choose to reuse the old one as base, or starting a newly implementation *(d)*. To use the old as base, the `base` keyword must be specified as `use` clause *(c)*. Inside an `extend` block, is possible to declare any new additional function, variable or implementation. However, is not possible to override existing public functions. Is also possible use extend, to implement an interface on a primitive type *(e)*. As constraint, all declared structures must be stateless. Any extension do not alter the type-safety. Even changing the implementation of some interfaces, the public semantic do not changes. Every dependence can be only at interface level, and all different implementations, share and expose the same public view. To block any extension, is possible to specify the `final` keyword. At class level, means that the class cannot be extended (either by adding or overriding an existing implementation). At implementation level, means that only that particular implementation cannot be overridden, and so for the function level. In addition to its protection purpose, `final` keyword allows the compiler to produce a more optimized code, reducing the amount of v-table request. If more than one module overrides the same interface of the same type, the order of the dependencies will establish the priority. Is possible to apply an extension of an existing type, only If no instances of that type has been created, before the extension module was loaded.

```
Public class File {                                      (e)
   public implement IInputStream {... }
   public implement ISeekable {... }
}
Public implement ITextWriter                             (c)
             for IInputStream {
   require local ISeekable;
}
public delay implement ILogger                           (b)
                   for ITextWriter {
  public function Log (msg : string,
                       level : LogLevel) {
     WriteLine(Now() + ":(" + level + ") " + msg);
  }
}
var x : = new File();
x.Log("Hello", LogLevel.Info");                          (d)
var y : ITextWriter = x;
y.Log("Hello2", LogLevel.Info"); //error                 (a)
```

A more powerful extension mechanism can be obtained using the implementation injection. With this technique, is possible to spread an interface implementation, in all context that implement the target interface. The instruction is "`implement` A `for` B", that means "if you implement B, then you will receive this implementation of A" . In the example, *(c)* say "if you are an `IInputStream` and also implement `ISeekable`, then you will inherit this implementation of `ITextWriter`". `File` class *(e)* satisfy those requirements, so automatically will be injected with this `ITextWriter` implementation. Now `File` is an `ITextWriter`. Similar, *(b)* say, "if you are a `ITextWriter`, then you will inherit this implementation of `ILogger`". Now `File` is also an `ILogger`, so is possible to invoke Log method from a `File` instance *(d)*. Similar for the local requests, all member of target interface will be accessible inside the `implement` block. Anyway, such extension will be applied only if the target context does not already implements this interface. To give the priority to the extension, must be used `override` modifier. Using the `delay` *(b)* clause, the extended interface will be loaded in target type, only after first access. This may fragment the heap, but avoids massive loading of rarely used services. The `implement for` statement does not alter the context and must be declared in global scope. If they was declared some local requirements, only the contexts that satisfy all constraints will inherit the service. If context is marked as `final`, then the extension will be ignored. Since that many types could not receive the extension, the access to the newly implemented interface is not allowed from target interface *(a)*. However, is possible to declare an `assert` constraint, to ensure that all target types implement the injected interface.

```
assert ITextWriter : ILogger
```

5.10 Event

An event in 'S is very similar to that one of c#, and can be defined as a notification mechanism to inform some actors, called listeners, that something is changed or happened. An event declaration is similar to a function declaration, except that an event cannot have a body, or return type. Events can be declared only at interface level, with keyword `event`. Only that one that implement this interface can raise that event, using `raise` keyword. Is possible to attach many listeners (also called *event handlers*). When an event is raised, all event handlers will be executed, in the order under which attach was made. There are two ways to attach an event handler. With `on` keyword, to declare a local-scoped event-handler. In this case, target event must be directly provided by a variable (or requirement) declared in the same (or parent) scope of event handler:

```
var car : Car;
on car.Move() { ... }                                    (a)
on car.Engine.Start() { ... } //error                    (b)
```

Regardless on declaration position, attach occurs any time that the event-source variable changes its value. Prior the execution of any attach, if the event handler is already attached to a different object, detach will occur. The detach will also occurs when either source object or event handler become out of scope. The event handler *(d)* must provide the same arguments declared in the event prototype *(e)*, except for the first one, that must be a reference to the interface in which the event is declared.

```
public interface IJob {
   event Started();                                     (e)
   function start();
   property Name : string;
}
public class Job {
   public constructor(name : string) {
      Name = name;
   }
   public implement IJob {
      public function Start() {
         raise Started(Now());
      }
   }
}
require IConsole;
public function Sample() {
   var j : IJob;
   j = new Job("job 1");                                (a)
   j.Start();
   j = new Job("job 2");                                (b)
   j.Start();
                                                        (c)
   on j.Stared(sender : IJob) {              (d)
      IConsole.WriteLine(sender.Name);
   }
}
//output
“Job 1” “Jbo2”
```

(a) event handler `j.Stared` is attached to the object `job 1`. *(b)* `j.Stared` is detached from the object `job 1`, and attached to the object `job 2`. *(c)* `j.Stared` is detached from the object `job 2`, because both `j` and `j.Stared` are out of scope.

When is necessary to let an event handler surviving, even when the source variable changes value or exit from the scope, is possible to invoke the `attach` and `detach` instructions.

```
function OnJobStarted(sender : IJob) { ... }   (a)
function Sample(jobs : Job[])
{
   foreach (var job in jobs)
     attach OnJobStarted to job.Started        (b)
}
```

`Attach` *(b)* accepts any function with the prototype compatible *(a)* to the event declaration. Also in this case, first argument must be a reference to the declaration interface. Using attach, there is no limitation on the event source member.

5.11 Generics

Generic types are implemented on the principle of those of C#, with similar syntax. Can be *generic* a class, an interface, a base or a filter. Currently, there is no support for covariance / contravariance, or for generic functions.

```
public interface IList<T> where T : IComparable {
    function Insert(item : T);
}
public base BaseList<T> : IList<T> where T { ... }
public base IntList : BaseList<int32> {... }

public class DoubleList<T> where T : IComparable {
   public implement IList<T> use BaseList<T> {
   }
   public implement IList<int32>
                use IntList
                 as SecondList {               (a)
   }
}
var x = new DubleList<string>();
x.Insert(“foo”);
x.Insert(1);
x.IList<string>.Insert(“foo2”);
x.SecondList.Insert(3);
var x = new DubleList<int32>(); //error        (b)
```

By `where` clause, is possible constrain the generic parameter to implement certain interfaces. Are also valid as constraint `value` (only a value type) or `class` (only a concrete class). For a better readability of the interface identifier, during an explicit member access, is possible to specify an alias *(a)*. Is possible to implement the same generic interface into the same class, providing differ generic arguments *(b)*.

5.12 Conditional Function

Is possible to declare two identical functions in the same scope, specifying a dispatch condition. Such functions must be marked with `conditional` keyword, and the condition can be expressed through `where` keyword. The condition can be any boolean expression evaluable in the function scope (the scope in which function is declared).

```
conditional function Format(
            value : int32
            format : string) : string
where (format == “x”) { ... }

conditional function Format(
            value : int32
            format : string) : string
where (format.StartsWith(“b”)) { ... }
```

If many functions satisfy the dispatch condition, only the first one will be called, in the order under which were declared. Once a function is marked as conditional, , all the instances of such function must be also conditional. A conditional function can’t be overridden, but can be overloaded.

5.13 Member Function

A global function can be mapped as a member of specific type or types. This can be useful when a function is not specific for a particular type, but works indistinctly with both, or to add some public shared function to an interface, that can be suitable regardless of the implementation.

```
function Print : ( printer : member IPrinter,
                  document : member IDocument,
                     pages : int32)
{ ... }
var x : IPrinter;
var y : IDocument;
x.Print(y, 1);
y.Print(x, 1)
```

Calling the function from the target type, the argument with its references will be hidden. More formally, if function *A* declares *(x, T1, T2, y)* arguments, and is mapped on type *T1* and *T2*, then *T1* will have *A(x, T2, y)*, and *T2* will have *A(x, T1, y)*.

If `member` modifier is specified in return type, the function can be used as constructor of specified type[11]. In this case, the function acts as a normal constructor, and `this` keyword refers to the instance of the newly created object:

[11] Only with class or struct type

```
function VerticalLine(pos: PointF,
                      len: float) : member Line {
   this.x1 = pos.x; this.y1 = pos.y;
   this.x2 = pos.x; this.y1 = pos.y + len;
}

function Origin() : member Point {
   x = 0; y = 0;
}

var l : Line;
l = new VerticalLine(new Origin(), 10);
```

`VerticalLine` is not a subtype of `Line`, is simply a named constructor. This technique can be used when constructor is ambiguous, and the behavior cannot be deduced by arguments. An hypothetical `HorizontalLine`, would be declared with the same arguments, but different meaning, and the standard constructor it could not be used.

5.14 Type conversion

By `conversion` keyword**,** Is possible to declare a user-defined type conversion. The scope in the conversion function, is that one of source type that must be converted. Converted value can be returned using `return` instruction *(a)*. A conversion, can be declared as `implicit` or `explicit`. Implicit conversion, will be automatically performed by compiler. Explicit conversion, instead, must be explicitly declared using the cast operator.

```
public struct Fraction
{
   public constructor(n : int32, d : int32) {
     Num = n; Den = d;
   }
   public var Num : int32;
   public var Den : int32;
}
public conversion implicit Fraction to float {
   return Num / Den;                             (a)
}
var f : float = new Fraction(10 / 5);
```

Declaring a conversion function, in fact it means defining a subtyping relation between involved types. Any high-cost conversion must be never declared as implicit, the programmer must be always aware of what he is going to do.

5.15 User defined type

By `type` keyword, is possible to define new types, based on an existing primitive type. This help to introduce different behavior associated to that particular data type, normally represented with a primitive type (e.g. a percentage, currency, color, etc).

```
public explicit type Email : string:           (d)
public type Currency : float
{
   public override implement IFormatter {      (e)
      public function Format() : string {
         return "€. " + Round(this, 2);
      }
   }
}
var x : Currency = 10.126;                               (a)
var y : float = x;                                       (b)
var z : object = x;                                      (c)
var k : Currency = y;                                    (f)
IConsole.Write(x); //€ 10.13
IConsole.Write(y); //10.126
IConsole.Write(z); //€ 10.13
```

However, we must always be aware of the meaning of this operation. A primitive type comes with no runtime type information. Casting a primitive type to an object (or any of its implemented interfaces), cause a box operation to be involved, so that any type information will be preserved *(c)*. Contrary, passing among two different primitive types, it cause the type-qualify to be loose *(b)*. An user defined type cannot change the physical nature of its base type, so is not possible declare any state information. However is possible to implement a stateless interface *(e)*. An user defined type is not strictly a subtype of its base type, both are interchangeable *(b)(f)*. However, due to the type loosing issue, is possible to specify the `explicit` qualifier during type definition *(d)*, so that only explicit cast will be permitted (except in literal assignation).

5.16 Conclusion

In conclusion, we want to propose our solution for the classic AST problem [61][63][65][31][37][45]:

```
public interface IEvaluable<T> {
   function Eval () : T;
}
public interface IExpression : IEvaluable<int>;
public interface ILeftRight : IExpression {
   property Left : IExpression;
   property Right : IExpression;
}
public interface ILiteral<T> : IExpression {
   property Value : T;
}
public class Literal {
   public implement IExpression;
   public implement ILiteral<int>;
   public implement IEvaluable<int> {
       public function Eval() : int {
          return Value;
       }
   }
}
public class Plus {
   public implement IExpression;
   public implement ILeftRight;
   public implement IEvaluable<int> {
       public function Eval() : int {
          return Right.Eval() + Left.Eval();
   }
}
```

With service model, we can take advantage of any existing implementation of any declared interfaces, even if in this example we do not use any base implementation of

`IExpression` or `IEvaluable`. A dynamic extension is possible even without source code:

```
assert IExpression : IFormattable;

public interface IFormattable {
   function Format() : string;
}
public extend class Literal {
   public implement IFormattable {
      public function Format() : string {
         return Value;
      }
   }
}
public extend class Plus {
   public implement IFormattable {
      public function Format() : string {
         return "(" + Right.Format() + "+"
                      Left.Format() + ")" )
      }
   }
}
```

6. RELATED WORKS

Our works cover mainly three topic (a) code reuse (b) modularity and extensibility (c) implementation independence. Most of related works, instead, cover only one of this aspects, so could be hard an effective comparison. With all trait[15][6], mixin[10][11][26][2][23], single / multiple inheritance, is not possible modify an existing type, any new feature can be declared only by defining a new type. Additionally the presence of many inheritable / reusable structures, laid a question: what must be expressed by means of traits or mixin? and what, instead, using specialization? In 'S classes cannot inherit, and is available an unique mechanism of code reuse. In 'S we can choose to apply an extension to an existing class, or to a family of classes (all that one implementing a certain interface), or even to define a new class, reusing the elementary building blocks that compose the old one. In both traits and mixins, the connection with the host class is performed by abstract methods, but different units may require a method with the same signature but different meaning. Additionally mixins must be linearly applied, and traits do not allow state. We will not argue anymore about this, the literature[47][37][31] is rich of more accurate analysis about the weakness of each of this methods. 'S does not require any mechanism to merge or intersect its base components, each interface defines a distinct behavior, and composition is linear. If two interfaces define a member with the same name, we can simply disambigue by specifying the name of the interface we are referring to. Aspects[44][39] are substantially different from services. They operate as observer, and provides a solution for problems that are cross-cut to the classes (es. Logging). Cannot be considered neither a unit of reuse nor a service, even if they can be used as they would be. Similar (but less powerful) results experienced with aspects, can be achieved using 'S filters. Existing languages do not provide a built-in support to realize the 'inversion of dependence'[38]. However, using some design pattern we can obtain similar results (for example, we could take advantage of c++ template and smart-pointers[1]). Multiple inheritance, suffers of diamond (duplicated base class). This may cause, duplication of state, problems during the initialization (multiple or ambiguous call to a base constructor), and the existence of multiple paths to reach a superclass method. Several technique was proposed to avoid this issues, such as linearization[3][17], renaming[40], virtual inheritance[18]. C++ virtual inheritance can handle any composition rules, without exceed on infrastructure. Base initialization problem expressed in [37], can be solved by using a specific init functions in place of the constructor. For instance, using the abstract class as interface, and expressing both the requirements and implementation by means of multiple inheritance, then we can obtain a composition power, comparable to that one of 'S. However, there would not be any formal distinction between what is a conceptual *requirement*, and what is instead an *extension*. In such situation, a class is allowed to "implement" an abstract function, even if its purpose was to be implemented elsewhere. CZ[37] suggests a solution that can resolve that ambiguity, by allowing to distinguish an extended class from a required one. Regarding single inheritance, a number of *design patterns*[27] (Visitor, Delegation, Observer, Abstract factory, etc) can be applied to obtain similar flexibility of 'S, however the pattern existence itself can be view as a lacks of target language: the inability to express a particular design problem with any built-in language construct. For the same reason we are using an OO language when we think "objects" (even such model can be fully expressed using 'patterns' in languages like C[56]), we should use a specific language to think "services". Our services model have no substantial difference with any existing *component model*. However, a *component model* covers a wide range of problematic, such as the interoperability between different languages, inter-process and network communication, that 'S does not cover. Compared to COM[54], 'S has the advantage to keep a local and dynamic interface-implementation catalog, instead of a global one. Additionally, has the capabilities to extend or change existing services, and to propagate an interface implementation, without edit source code. A method-based extensibility similar to 'S *member functions* is available also in Open classes[13], expander[64], and extension methods[41]. All adopt similar approach, and all suffer of the inability to declare state. Multijava[13] and CZ[37] languages, provides a mechanism for dynamic dispatching, similar to conditional function of 'S. Nevertheless, 'S allow to express the dispatch condition by using expressions that can evaluate even external conditions. Other approaches to extensibility are virtual classes[35], nested inheritance[45], or scala[49] abstract type. All this technique allow a massive extension in a set of correlated and nested classes (family polymorphism[20]). One advantage is that each extension lays under a special class that “host” all extendible classes. Any extension is not invasive, and we can always choose to use the old class family instead of the extended one. The most notable model diverging from classic OO, is that one prototype-based. First in self [62] and then other languages, such as ECMAScript[25], the prototypes offer a natural way to think objects. All is an object, and all object instances are created by cloning an existing object (the prototype). Newly created object, can diverge from its original copy and define new behavior or state. However, that imply a dynamic type system. The world of dynamic / *duck typed* (e.g. ruby [24]) languages, allow a flexibility and expressiveness that many static and strongly typed approach does not. However, we will not make any comparasion with such languages, because simply are based on different philosophies. ‘S is born to be fast. A dynamic system cannot be realized without a cost. Most of the member access must be dynamically dispatched using a name dictionary, and the memory slot holding a dynamic typed object at runtime, must contain all the structure necessary to self-describe the type, and even, to modify it. Over the performance issues, there is a number of advantages in static / strongly typing, such as reduce runtime errors, simplify the debug and coding, help the development tools to introduce more facilities (e.g. Intellisense). A static-typed prototype-based system was advanced in [31]. In such model, there is no more distinction between objects and types. All is an object (or a type), even the literals are cloned from its base type (e.g. ‘3’ is subtype of int). New objects can be defined, also by combining more than one prototype. Similar to ‘S, prototypes provides a simple and unique mechanism for code reusing, and maybe they can be considered the most valid alternative to our approach.

7. TRANSLATION TO C#

We do not provide in this paper any formal definition (either grammar or semantic) of ‘S, however we show how the type system of 'S can be full expressed using the type system of c#.

7.1 Overview

Some features have a direct equivalent in C#, some other requires some support code provided by our runtime library[29]. Use of such library, however, must be integrated with a lot of infrastructural code, that anyway may be auto-generated (in a future) using a translation tools.

C# supports variables, properties, events, structs, type conversions, exceptions, interfaces and interfaces inheritance in the same way ‘s does. C# 4.0[41] also supports type inference, and auto-implemented properties.

7.2 Function

C# 3.0 does not support optional arguments (will be supported in c# 4.0). Alternatively is possible to use method overload:

```
//s
function format(format : string = “”) {...}

//c#
void Format() {  Format(“”) }
void Format(string format) { ... }
```

C# does not support code outside a class context, thus any global function must be declared as static into an utility class. Extension methods may be used to group these functions into a “virtual type”.

7.3 Context

Context is emulated with a static class called `Context`. This class manages the context-stack, and allow a quick access to local and global context (respectively with Local and Global property). A call to `Enter` method will cause specific context to be pushed into the stack, a call to `Exit` to be popped. Any call to enter / exit, must be performed inside a `try - finally` block, to ensure that context will be restored properly, in case of exceptions. Is possible to enclose any context specifics code in a `using` block, passing a wrapper object (`ContextScope`). This object will invoke automatically `Context.Enter` (using the context passed to the constructor), and then `Context.Exit` in `Dipose` method.

```
//s
enter MyContext {
   ...
}

//c# (a)
Context.Enter(MyContext.Instance);
try {
   ...
}
finally {
   Context.Exit(MyContext.Instance);
}
//c# (b)
using (new ContextScope(MyContext.Instance)) {
   ...
}
```

All classes representing a context, must implement `IContext`. Since a service can be implemented only inside a context, a class implementing a service must also implement `IContext`, so every service interface can extends `IContext`.

```
public interface IContext {
   T Require<T>() where
             T : class, IContext;
   void Implement<T>(IContext obj) where
                  T : class, IContext;
   IContext Parent { get; set; }
}
```

Calling `Implement<T>` is possible to assign a `T` service implementation on that context. `Implement<T>` can has different behaviors depending on context type. Every call to `Implement<T>` or `Require<T>` on `Context`, will be redirect to the current (local) context.

```
//'s
implement ITextWriter use new Console()
require ITextWriter;
ITextWriter.WriteLine("Hello");

//c#
Context.Implement<ITextWriter>(new Console());
...
Context.Require<ITextWriter>().WriteLine("Hello");
```

7.4 Implementation

Any 'S-interface-implementation must be performed in a separated class, inheriting from `Base<T>` (where `T` represent host context, the context in which implementation is used). Host context is passed on constructor, and then stored in a private field (of `T` type) called `_instance`. Parameter `T` must be specialized only in classes and in user-defined contexts, bases must remain generics on `T`. `Base<T>` implements `IContext`, redirecting any request to host context. Local requirement can be represented using a generic-argument constraint on `T`, so that will be also possible have access to all required interfaces using `_instance` without any cast or runtime type-checks. Not-local requirements, must be declared in specific fields (of the same type of required interface), and initialized calling `Contex.Require`:

```
//'s
public base BaseStream : IStream {
    require local ISeekable;
    require IConsole;
}

//c#
public class BaseStream<T> : Base<T>, IStream
                   where T : ISeekable {

    protected readonly IConsole IConsole
            = Context.Require<IConsole>();

    public BaseStream(T instance)
           : base (instance) { }
}
```

Unimplemented interface members must be declared as abstract (and therefore, the base class too). If implementation handles an interface that uses inheritance, all inherited interfaces must be treat as local requirements, and declared as constraints of T. Then, such interfaces must be `implicit` implemented, redirecting each method to the host class

```
//'s
public interface IStream { function Close(); }
public interface IInputStream : IStream { ... }
public base BaseInputStream : IInputStream { ... }

//c#
public class BaseStream<T> : Base<T>, IInputStream
                   where T : IStream { ...
    void IStream.Close() {
        _instance.Close();
    }
}
```

7.5 Class

Any 'S-class must inherit from `ServiceObject`. Every implemented interfaces, must be declared into a private nested class that inherits from `Base<T>` or from an existing base. `T` argument must be specialized with the class type. For argument must be specialized with the class type. For each implemented interfaces, must be declared also a specific private field that will hold the implementation instance, and a public property with the same name / type of implemented interface. This property can be used to have a quick access to ambiguous members. Finally the class must implement all interfaces, redirecting every call to associated implementation instance. All conflicting members must be implemented as implicit.

```
//'s
public class File {
   public implement IStream {
      public function Close() { ... }
   }

   public implement IInputStream use
                    BaseInputStream;
}

//c#
public class File : ServiceObject, IInputStream {
   class Stream : Base<File> : IStream  {
       public void Close() { ... }
   }
   class InputStream : BaseInputStream<File> {
   }

   Stream _stream = new Stream(this);
   InputStream _inputStream =
                      new InputStream(this);

   public IStream IStream {
      get { return _stream; }
   }
   public IInputStream IInputStream {
      get { return _inputStream; }
   }

   public void Close() {
      _stream.Close();
   }

   public int ReadByte() {
      return _inputStream.ReadByte();
   }
}
```

Equivalent to “S”, we can have access to implemented service, using a flattern or component-driver view:

```
File file = new File();
file.ReadByte();
file.IInputStream.ReadByte();
file.Require<IInputStream>().ReadByte();
(file as IInputStream).ReadByte();
```

7.6 User defined type

User defined type can be emulated using structs and conversion operators:

```
//'s
public type Percentage : float;

//c#
public struct Percentage {
   float _value;

   public Percentage (float value) {
      _value = value;
   }

   public static implicit
                operator float(Percentage obj) {
      return obj._value;
   }

   public static implicit
                operator Percentage(float value){
      return new Percentage(value);
   }
}
```

7.7 Implement for, and class extension

The role of `ServiceObject` is also manage dynamic extension . Each extension will be added to an *internal* list of `RuntimeExtension`:

```
public class RuntimeExtension  {
   public object Instance;
   public Type   ExtensionType;
   public IEnumerable<Type> ImplementedServices
}
```

Upon the first call to `Request` or `Implement` methods, will be enumerated all suitable extensions for that class, even those that are referred to its services. The extension must be implement into a class inheriting from `Extender<T>`, (where `T` is the extension target) and registered to the extensions catalog, held by the Context:

```
//'s
public extend class Stream {
   public implement ILogger { ... }
}

//c#
public class StreamExtender : Extender<Stream>,
                              ILogger { ... }
Context.ExtendBy<StreamExtender>();
```

The extension class will be created only after the first request to one of its services. After creation, extension class will receive the instance of extended class, calling `Bind` method. `Extender<T>` implements `IContext`, all requests made to an unimplemented service, will be redirected to the target object.

```
Stream obj = new Stream();
ILogger logger = obj.Require<ILogger>();
IStream stream = logger.Require<IStream>();
```

`ILogger` is implemented through an extension class, therefore `logger` variable is a `StreamExtender` instance. `IStream` is not implemented by `StreamExtender`, therefore request handling will pass to extended object (`Stream`), that can successfully satisfy it.

Using extensions methods, is possible to flattern an extension interface:

```
public static ILoggerForIStreamExtender {
   static string Log(this IStream stream,
                     string text) {
      return stream
             .Require<ILogger>()
             .Log(text);
   }
}
var obj = new Stream();
obj.Log();
```

7.8 Conditional function

Conditional function are not easy to emulate, and needs a lot of infrastructure. Every conditional function must be represented by an interface with two methods: (a) `Check`, used to evaluate the condition, (b) `Invoke`, used to realize the function. Both methods must accept as first argument the instance of the type that declare conditional function, and then all arguments of target function. Every condition / body must be implemented in a distinct class. Finally, must be declared a static class holding all the instances of varies implementations, exposing an `Invoke` method that will dispatch the call to the first one that will satisfy the condition:

```
//'s
conditional function FormatHex(string : format)
         : string where (...) { ... }
conditional function FormatBinary(string : format)
         : string where (...) { ... }
//c#
public interface IFormatFunction {
   bool Check(int32 target, string format);
   string Invoke(int32 target, string format);
}

public static class FormatFunction {
   static List<IFormatFunction> _list;

   public static void Add(IFormatFunction item) {
     _list.Add(item);
   }

   public static string Invoke(this int32 target,
                      string format) {
     foreach (var item in _list)
       if (item.Check(target, format))
          return item.Invoke(target, format);
   }
}

class FormatHex : IFormatFunction { ... }
class FormatBinary : IFormatFunction { ... }

public static void Main() {
   FormatFunction.Add(new FormatHex());
   FormatFunction.Add(new FormatBinary());
}
```

7.9 User defined context

Any 'S-user-defined context must inherit from `UserContext`, and have only a single shared instance accessible in a static property called `Instance` (singleton). Every implemented services must be declared on constructor calling `Implement<>` method:

```
//'s
public context CommandLine {
   public implement ITextWriter use Console;
}

//c#
public class CommandLine : UserContext {
   private CommandLine() {
      Implement<ITextWriter>(new Console());
   }
   public static readonly CommandLine Instance =
               new CommandLine();
}
```

7.10 Member-mapped function

Single-type member function can be mapped one-to-one with extension methods. Else must be declared one extension method for each target member, that must appear as first argument:

```
//'s
function Print(printer : member IPrinter,
               document : member IDocument,
               count : int32) { ... }

//c#
public static class PrintExtender {
   static void Print(this IPrinter printer,
                     IDocument document,
                     int copyCount) {
      ...
   }

   static void Print(this IDocument document,
                     IPrinter printer,
                     int copyCount) {
       Print(printer, document, copyCount);
   }
}
```

7.11 Custom constructor

The custom constructors cannot be realized. Alternatively, can be used a static functions in target class (or utility class, if source code is not available).

```
//'s
public function Origin() : member Point {
   x = 0; y = 0;
}
var p : Point = new Origin();

//c#
public class Point { ...
   public static Point CreateOrigin() {
      return new Point() { x = 0, y = 0 };
   }
}
Point p = Point.CreateOrigin();
```

8. CONCLUSION AND FUTURE WORK

We have shown an alternative way to think and model the world of objects, that does not suffer of many of the limitations, found in related works. 'S is unambiguous, and provides an essential, but powerful set of composition structures. 'S simplify the components creation, allowing a widely reuse mechanism and a better maintenance, without loose the advantages of a static and strongly type system. Even if without a rigorous and formal translation, we have shown how 'S can be fully mapped in a traditional OO language, such as C#. This let we hope that some formal properties proof in C#, will be also valid 'S. However, the next step on 'S development, is write a formal definition. Generate c# has several advantages, first of all, have access to the huge class library provided by .NET Framework. Nevertheless, the first aim of 'S is to be a decomposable system, that can run even in a low-resources environment. Thus, we are currently working to an 'S compiler that can generate pure C code. C means portability and speed in almost any platform. Due to the synthesis needs and an immature developing status, we decided to not discuss about it in this paper. As anticipation, we are experimenting in our test, a very effective code, needing a minimal runtime environment. This let us to hope that an 'S program will be run also in a embedded system.

9. ACKNOWLEDGMENT

(todo)

10. REFERENCE

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001.

[2] D. Ancona, G. Lagorio, and E. Zucca. Jam - designing a Java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003.

[3] K. Barrett et al. A monotonic superclass linearization for dylan. *In OOPSLA (1996)*, pages 69-82.

[4] D. Batory, J. Liu, and J. Sarvela. Refinements and multi-dimensional separation of concerns. *ACM SIGSOFT*, 2003.

[5] D. Batory, J. N. Sarvela, A. Rauschmayer. Scaling step-wise refinement. *In ICSE (2003)*, pages 187-197

[6] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[7] L. Bergmans. The composition filters object model. In *Proceedings of the RICOT symposium on Enabling Objects for Industry*, 1994.

[8] A. P. Black, N. Scharli. Traits, Tools and Methodology. *In ICSE (2004)*, pages: 676 - 686.

[9] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA (1997)*, pages 66–76.

[10] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP*, 1990.

[11] G. Bracha. *The Programming Language Jigsaw*: *Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, University of Utah, 1992.

[12] Y. Caseau. Efficient handling of multiple inheritance hierarchies. *In OOPSLA (1993)*, pages 271-287.

[13] C. Clifton, G. T. Leavens, C. Chambers, and T. *Millstein*. MultiJava: modular open classes and symmetric multiple dispatch for Java. *In OOPSLA (2000)*, pages 130–145

[14] F. R. Campognoni. IBM's system object model. *Dr. Dobb's*, pages 24–28, 1994. Winter 1994/1995.

[15] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A.P. Black. Traits: A mechanism for fine-grained reuse. *In ECOOP*, 2003

[16] M. Dooren, E. Steegmans. Language constructs for improving reusability in object-oriented software. *In OOPSLA (2006)*, pages 118-119.

[17] R. Ducournau, M. Habib, M. Huchard. M. L. Mugnier. Proposal for a monotonic multiple inheritance linearization. *In OOPSLA (1994)*, pages 164-175.

[18] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[19] E. Ernst. Propagating class and method combination. In *ECOOP*, 1999.

[20] E. Ernst. Family polymorphism. *In ECOOP*, 2001.

[21] E. Ernst. Higher order hierarchies. *In ECOOP*, 2003

[22] E. Ernst, K. Ostermann, W. R. Cook. A virtual class calculus. *In POPL (2006)*, pages 270–282.

[23] R.B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. *ACM SIGPLAN Notices*, 34(1):94–104, 1999.

[24] D. Flanagan, Y. Matsumoto. *The ruby programming language*. O'Reilly, 2008.

[25] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media. Inc.. 2006

[26] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL '98*, 1998.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[28] S. Gudmundson, G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *Advanced Separation of Concerns, Workshop at ECOOP* (2001).

[29] A. Guerrieri. *Service Model C# Library*. http://www.eusoft.net/serviceModel.rar. 2010.

[30] W.H. Harrison, H. Ossher. Subject-Oriented Programming (A critique of pure objects). *In OOPSLA (1993)*, pages 411–428.

[31] D. Hutchins. Eliminating distinctions of class, using prototypes to model virtual classes. *In OOPSLA (2006)*, pages 1-20.

[32] N. Josuttis. *Soa in Practice: The Art of Distributed System Design*, O'Reilly Media, Inc., 2007.

[33] G. Kiczales, M. Mezini. Aspect-oriented programming and modular reasoning. *In ICSE (2005)*, pages 49–58.

[34] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. *In ECOOP*, 2005.

[35] O. L. Madsen, B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. *In OOPSLA (1989)*, pages 397 - 406

[36] O. L. Madsen, B. Møller-Pedersen, K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[37] D. Malayeri, J. Aldrich. CZ: multiple inheritance without diamonds. *In OOPSLA (2009)*, pages 21-40.

[38] R. C. Martin: The Dependency Inversion Principle. In *The C++ Report*, 1996.

[39] H. Masuhara, G. Kiczales. Modeling crosscutting in aspect oriented mechanisms. *In ECOOP (2003)*, pages 2–28.

[40] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[41] Microsoft Corporation. *C# Version 4.0 Specification*. http://download.microsoft.com/download/7/E/6/7E6A548C-9C20-4C80-B3B8-

860FAF20887A/CSharp%204.0%20Specification.doc. March 2009.

[42] L. Mikhajlov, E.Sekerinski. A Study of the fragile base class problem. *In ECOOP (1998)*, pages 355–382.

[43] D. A. Moon. Symbolics Object-oriented programming with flavors. *In OOPSLA (1986)*, pages 1-8.

[44] G. Murphy, C. Schwanninger. Aspect-oriented programming. *IEEE Software 23:1 (2006)*, pages 20–23.

[45] N. Nystrom, S. Chong, and A. Myers. Scalable extensibility via nested inheritance. In *OOPSLA (2004)*, pages 99–115.

[46] N. Nystrom, X. Qi, and A.Myers. J&: Nested intersection for scalable software composition. *In OOPSLA (2006)*, pages 21-36.

[47] N. Nystrom. *Programming Languages for Scalable Software Extension and Composition*. Ph.D. thesis, Cornell University, 2007.

[48] M. Odersky and M. Zenger. Scalable Component Abstractions. *In OOPSLA (2005)*, pages 41-57.

[49] M. Odersky. *The Scala Language Specification*. At: www.scala-lang.org/docu/files/ScalaReference.pdf Version 2.7. March 2009

[50] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[51] H. Ossher, P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. *Proc. Symp. Sw. Arch. & Component Technology*, 2000.

[52] A. Paepcke. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993

[53] W. H. Peri Tarr, H. Ossher. N degrees of separation: Multidimensional separation of concerns. *Proceedings of ICSE*, 1999.

[54] D. Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, 1997.

[55] M. Sakkinen. Disciplined inheritance. In ECOOP, pages 39–56, 1989.

[56] A.T. Schreiner. *Object-Oriented Programming With ANSI-C*. At http://www.planetpdf.com/codecuts/pdfs/ooc.pdf. 1999

[57] G. Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, 1994.

[58] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA (1986)*, pages 38–45.

[59] F. Steimann. The paradoxical success of aspect-oriented programming, *In OOPSLA'06*, pages 481-497, 2006

[60] L. A. Stein. Delegation is inheritance. *In OOPSLA (1987)*, pages 138-146.

[61] M. Torgersen. The expression problem revisited. four new solutions using generics. *Proceedings of ECOOP*, 2004.

[62] D. Ungar and R. Smith. Self, the power of simplicity. *Proceedings of OOPSLA*, 1987.

[63] P. Wadler et al. The expression problem. *Discussion on Java-Genericity mailing list*. December 1998.

[64] A. Warth, M. Stanojević, T. Millstein. Statically scoped object adaptation with expanders. *In OOPSLA (2006)*, pages 37-56.

[65] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. *Workshop on Foundations of Object-Oriented Languages*, 2005.