Service-Oriented Programming – Supporting Emerging Variations of Object-Oriented Programming

William Harrison, Tim Walsh

Department of Computer Science
Trinity College
Dublin 2, Ireland
(+353) 1-896 8556
{Bill.Harrison | Tim.Walsh}@cs.tcd.ie

* This research is supported by a grant from the Science Foundation Ireland.

© William Harrison, Tim Walsh, 2006. Permission to make digital or hard copies of all or part of this work for personal use is granted provided that that copies bear this notice and the full citation on the first page.
ABSTRACT
Much of the object-oriented software produced today is written in programming languages in which clients request services from specific objects. But important areas of software development, like aspect-oriented, pervasive, service-oriented, grid, and event-driven, require a more flexible model for how services are found. We describe here a re-interpretation of familiar object-oriented concepts that enables object-oriented software to be much less dependent on the structure of the implementations of the services that they call. This new interpretation of the concepts of object, reference, and method call is intentionally structured to extend the existing interpretation, allowing existing software to be carried forward compatibly and even to exploit some of the flexibility it introduces. We outline important features of a new programming language, Continuum, under development by the Software Structure Group at Trinity College Dublin, that is intended to more directly address needs of these communities.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – abstract data types, polymorphism, control structures.

General Terms
Languages

Keywords
Object-Oriented, Aspect-Oriented, Service-Oriented, Pervasive, Symmetric Multimethods, Context, Dispatch, Garbage Collection

1. OVERVIEW
The environment in which we create and use software and the trade-offs between developer resources and machine resources has changed considerably since use of the common model of object-oriented software development became widespread in the early 1990’s. There are indications that it is time to deploy an alternative approach to the use and deployment of objects that better satisfies the emerging needs. Section 2 of this paper summarizes emerging variations of the common approach to object-oriented that indicate a change is needed. Section 3 summarizes the common issues that lie under these emerging variations. Section 4 outlines how these issues can be addressed by loosening the constraints on how service clients’ calls for service, and how these calls are addressed by service providers. It introduces Continuum as a programming language providing full access to the flexibility available with the looser constraints. Section 5 then outlines various degrees by which the looser constraints can be applied to existing and new software.

The Software Structure Group at Trinity College Dublin is investigating programming-language solutions that allow client software to be less dependent on the object-structure adopted by service providers, focusing more on the services themselves than on the objects that implement them. We can call such a language a “Service-Oriented Programming” language to emphasize this change in focus.

2. CLUES TO THE NEED FOR CHANGE
Object-oriented software is usually characterized by the joint emphasis of inheritance, encapsulation, and polymorphism. But while admitting occasional exceptions, such as CLOS (the Common Lisp Object System) [9] and Cecil [4], the most common model for object-oriented software is that in which a method is implemented in the object identified particular reference at the point of call, called the “target” of the call, or which, in many systems, the message is sent to the “target” object which provides its implementation. This common interpretation is breaking down in the modern computing environment, and we can see symptoms of this breakdown emerging in a wide spectrum of variations, including aspect-oriented software [6], pervasive or autonomic computing [18], service-oriented software [17], and “Grid” computing [16]. These models all share at least one important difference from the common OO model: they interpose or presume a dispatching intelligence between the message-sending client and the message-receiving servicer. While the emergence of these new models clearly highlights the need for such a dispatch intermediary, we can also see how its introduction would help to simplify the ongoing attempts to allow the separation of business logic from contextual concerns.

* This research is supported by a grant from the Science Foundation Ireland.
2.1 Aspect-oriented Software
Aspect-oriented software development embodies two characteristics: interception of the behavior of a body of software at points that may be unexpected by the original software writer, and application of behavior defined separately in “aspects” before, after or around the behavior originally expected at those points. The original “message” is not sent directly to the target object. In fact, depending on the aspects’ definitions, the target object may never see the message at all.

2.2 Ubiquitous/Pervasive Computing
One of the key characteristics of pervasive computing is its use of context to find implementations of services that are needed by clients. The contextual choices may involve location, signal strength, reliability or other information generally unknown to the clients or servers themselves. The complexity of these applications is often significantly increased by the use and management of proxies that implement the context-sensitive parts of the infrastructure, separating the apparent target from a contextually-selected real target.

2.3 Service-Oriented Software
Service-oriented architectures accommodate the registration, discovery, and use of services that provide a higher-order orchestration of methods. The methods would conventionally be applied to many different kinds of objects manipulated by the service internally or even referenced by the client. The client applications generally originate and send messages to an object provide as the service’s “gateway”. The gateway can accommodate transaction history and other context information, in deciding how to route the interaction to the “real” implementation objects it provides for the references.

2.4 Grid Computing
Scientific applications form a large part of the software of interest to the Grid computing community, and not all of it is written in an object-oriented style. But the issue of the distribution of calls on a method or function among a variable and even, in fact multiple, set of servicing receivers is fundamental to the issues being faced by that community. What makes the Grid work is the fact that, as with pervasive computing, the client programs don’t have to be aware of the availability and number of service elements. When phased as objects, requests for service are often addressed to a dispatching object that uses information about processor availability and power to subdivide the requests, for delivery to the actual objects that will service them.

2.5 Event-driven Software
The computing model based on the reliable forwarding of asynchronous events from client to service is important to many application communities because of their distribution world-wide and because the set of parties interested or needed to handle the processing varies strongly over time. The fact that the client and the server need not even be available at the same time has great advantage to enterprises with offices in both New York and Hong Kong, and the ability to subscribe to published events allows for the creation of an economy based on event-passing rather than on direct communication.

2.6 Separating Business Logic from Context
One characteristic of software that is important to the commercial world today is the ability to rapidly adapt its structure to changing organizational and operating environments. Where merger, acquisition, and unbundling are key characteristics of the business environment, there is a critical need to be able to use clients and services from one part of an organization in conjunction with those of others. Software in which clients is aware of the structure of the service providers is too brittle to be redepolyed in this manner. Businesses critically need to avoid mingling the logic that manages the persistence, transaction, or service-structure context of a client with the logic of its business processes. One step toward enabling the rapid reorganization of software and toward increasing its malleability was the introduction of J2EE/Enterprise Java Beans [20] and ‘.net’ components [21]. They provide an incomplete approach to separating the operating context from the business logic of software. While separating out concerns like data storage and transaction management and achieving some of the desired malleability, the resulting software has proved to be more complex than expected. This is due, in part, to the need to explicitly find and manipulate the objects representing the rewrapped business logic. These early approaches to separating business logic from operating context are being displaced in many contexts by newer variations of the object-oriented model. Continual reorganization is one of the motivators for the growing popularity of both aspect-oriented technologies to and service-oriented technologies.

3. SUMMARY OF THE COMMON ISSUES
When we review these many emerging variations of the object-oriented model being put forward for software, we can see several issues shared among them. This section summarizes the common issues and argues that we need to change the way we think about method calls from the client’s point-of-view. The next section will then present a foundation for thinking about a service-oriented programming language that can resolve these issues.

3.1 Applying Context to Message Dispatch
All of these modern variations of the object model interpose a dispatching intelligence between the message-sending client and the message-receiving service. This dispatching intelligence uses the “target object” of the method call as only one of the pieces of information identifying the real target to which the message is delivered. The role of the dispatching intelligence is to take account of the environmental context: aspects, locality, service-selection, available processors, or manner of deployment. Doing so may involve examining both information presented explicitly in the call and information known implicitly about the client and the potential services.

3.2 Calling Services Instead of Objects
Most of these modern variations of the object model suffer from having their actualizations forced into the common model which is the only one made available by the programming languages in use. Some variations suffer from the need to manage proxies for the targets. Other variations treat the dispatching component as the target object itself. But in all cases, their ability to successfully address the needs of their communities is limited by the inability of a client to indicate “I don’t know which object implements the service.”. In all cases, they would be better served by a model in which the client focused on calling a method
without specifying which object was expected to implement the method.

3.3 Client View vs. Supplier View
Changing the client’s emphasis from “sending a message to an object” to “calling for a service” need not destroy the concept of object, or of object-oriented software. Objects are described well as built using inheritance with encapsulated implementations selected polymorphically by the dispatcher. Classes provide good way for suppliers to capture the interdependent specification of state representation and behavioral specification. But there is no need that the objects implementing services be the focus of the clients using them. An object may have the view that it is the (sole) provider of a service. But the client need not share this view and may simply indicate a need for the service. The dispatching component is responsible for matching a call for service to a servicer, or perhaps to many servicers.

The widespread presence of common themes in the suite of variations argues that it is time to widen the common model to accommodate the fact that clients are service-oriented, and that objects are a secondary issue to them.

3.4 Non-disruptive Change
While it is important to identify a foundation for software that addresses the needs evidenced by these emerging variations, the huge investments already made in both people and artifacts must be conserved. It is not actually necessary to sacrifice this investment in providing a new language for expressing software. We can place a new language on a foundation that can be laid under the existing software, and we can reinterpret the existing software in the light of this foundation. While legacy software will not acquire all of the malleability of new software using the new language, the investment can be conserved, and some additional malleability can still be achieved.

4. RESOLVING THE ISSUES
We have identified a need to change the programming model to accommodate four issues: applying context to message dispatch, calling services instead of objects, supporting a client view that differs from the servicer’s view, and introducing change in a non-disruptive manner. A change in language is needed because accommodating the malleability needed in modern software requires that the artifacts themselves be malleable. This is not possible if too much information about context is embedded in the executable logic of the software. The most popular object-oriented languages irremediably fail to address this need, and it is time to lay the groundwork for a programming language or a kind of programming language that focuses more on services than on objects.

The object-oriented software paradigm is particularly strong because it rests on several mutually-supporting legs: objects implement state and behavior, reference values point to objects, and reference variables are typed by interfaces expressing capabilities needed. Not surprisingly, therefore, any new foundation is also likely to rest on several mutually-supporting changes. The reasons for elements of the change may arise directly from one of the issues describes in the previous section, or indirectly, in response to one of the direct elements.

Since it is not practical to address all elements of a solution first, we begin with an outline that can be used to illustrate how the elements of the language work together and why they are introduced. In Section 4.1 we focus on supporting the client’s change from calling objects to services, issue 2 from above. This discussion comes in 5 parts, using bold-italic font to highlight terms that can be read informally at the moment, but will be given particular technical meanings in the section where they are introduced:

1. First we provide a discussion of symmetries and how breaking them reduces malleability and observe how two important symmetries are broken in the common model.
2. To reintroduce one of the lost symmetries we employ the concepts of assurances and reconbinance, and we discuss the issue of how necessity relates to reconbinance.
3. We then describe a small example to illustrate assurances, reconbinance, and necessity, setting them in the familiar context of interfaces and classes.
4. To reintroduce the second of the lost symmetries we employ the concept of generalized references which are fully typed references that can be represented as computational values other than pointers.
5. The need for generalized references leads into the introduction of the miniature construct – a generalization of the identity creation function commonly built-in to languages.
6. We can then summarize the information that must be provided in a parameter declaration and describe how to generalize the concept of interface into that of an outerface to provide the type-model for this family of languages.
7. We then revisit the small example illustrating it with the extended syntax for generalized references.

Then, in Section 4.2, we discuss use of context in dispatch, issue 1 from above. In any viable software deployed today, the dispatch component must accept clients and services written in many paradigms and languages. So it can not properly be described as part of the Continuum language. Instead, the role of the dispatcher is discussed, and the need for its interaction with clients, services, and other dispatchers.

Finally, in Section 4.3, we address the concepts involved in the servicer’s view, separated from the client’s as issue 3 from above. The client’s view has defined a new type-model which has implications for the interpretation of classes and modules in the servicer. Services are a linguistic element in Continuum. They provide the resources management usually that is built-in to languages for pointer references, and they assure that the method implementations in the classes and modules using generalized references can directly access the underlying state they are defined to embody – an activity called decapsulation. Although context plays a role in selecting a service, dispatch also occurs within a service to find the appropriate method. For services implemented in Continuum, we indicate how specialization different from or inheritance, and how ambiguity is treated. We then revisit yet again the small example, this time focusing on the service-side constructs that come into play in making it work.

4.1 Calling Services Instead of Objects
4.1.1 Symmetries
In Physics, Mathematics, and other branches of science the concept of symmetry plays an important role. Symmetry characterizes systems in which the interchange of variables leaves the systems unchanged. For example, squares are symmetric when
their coordinates are exchanged about any of 4 axes through their center, but circles are symmetric about any axis through theirs. Greater symmetry renders shapes that are more interchangeable and useful in different placements. Subatomic particle reactions produce particles in a manner symmetric with respect to a number of conserved parameters, and only the fact that the universe has cooled considerably has rendered some of these interchanges no longer practical. Generally speaking, the breaking of symmetry reduces the number of degrees of freedom in a system. The same is true in programming language design. The formulation of method-call in languages supporting object-oriented programming today breaks two important symmetries, which we can call method-parameter symmetry and key-reference symmetry.

Method-parameter symmetry characterizes systems in which type-checking gives no special role to any one of the parameters over the others. Languages supporting the common approach to object-oriented programming break this symmetry by identifying a syntactically and semantically distinguished parameter, called the “target”, whose type guarantees methods that can be called through it. Breaking the symmetry has the effect that even if an unexpected implementation for the method exists in a non-target object, that implementation cannot be used. This renders the client dependent on the overall structure of the objects providing services.

Key-reference symmetry characterizes systems in which typing information about the capabilities for calling methods is associated only with references that have a particular representation, such as a pointer. In most object-oriented languages, computational data such as strings containing bar-codes must be used to find a pointer-based reference to an object before typing information defining the capability to call methods can be known or verified. Breaking the symmetry has the effect of forcing a client to use the key to find such an object. This introduces into the client complex code determining where to find the object, which code depends on the structure of the supporting service’s implementation.

4.1.2 Restoring Method-parameter Symmetry

Method-parameter symmetry requires that the client be unaware of where the implementation resides – of which parameter’s type provides the assurance that the method can be called safely. Considering only interfaces for the moment, we will first define several concepts (in italic bold-face) here, giving them the meaning they commonly carry, although phrased in terms we will use to redefine some of these concepts shortly.

Reference variables are declared to have types specified as interfaces, so a variable declaration associates a variable \( p_0 \) with an interface \( I_0 \).

An interface is a set of methods that can be called given a non-null pointer variable that is declared using that interface. The method is generally specified using a name and a series of interfaces that specify types for its parameters. For later convenience, we can model them as a series of signatures each of which has the form \( m(I_0, I_1, \ldots, I_n) \) which must be satisfied by the target of the method call. These signatures are called the assurances given by the interface. The method name is \( m \), and the \( I_i \) are the parameter interfaces. \( I_0 \) is a unique single-method interface \( m(I_0, I_1, \ldots, I_n) \).

With these as a base, we make three others that will change:

1. Interfaces usually form a lattice defining subtypes. An interface \( J \) is a subtype of an interface \( I \) iff it assures everything in the supertype, that is, each of the signatures in \( I \) conform to at least one signature in \( J \), and \( J \) is explicitly declared to be a subtype of \( I \). The requirement that it be explicitly declared as a subtype is used with type systems that can be called “nominal.” With nominal subtyping the subtyping is not only derived from structure, but must also be pronounced by name.

2. Conformance is used when comparing types in parameter positions of signatures, to accommodate the concept of subtyping. A signature \( m(I_0, I_1, \ldots, I_n) \) conforms to a signature \( h(J_0, J_1, \ldots, J_k) \) iff \( m=h \), \( n=k \), \( J_i = I_i \) and for all \( 1 \leq i \leq n \), \( I_i \) is a subtype of \( J_i \). The equality condition must be used when comparing the target parameter of the signature because nominal subtyping cannot be applied. It cannot be applied because the developer may legitimately have several differently named interfaces \( K \) each containing only \( m(K, J_0, \ldots, J_n) \).

3. We say method call, written \( p_0.m(p_1, \ldots, p_n) \) is assured if \( p_0 \) is not null and \( m(p_0, p_1, \ldots, p_n) \) conforms to a signature in the interface \( P_0 \) declared for the variable \( p_0 \).

As we noted above, these definitions obviously break the symmetry among the parameters because of the formulation of rules 2 and 3. More subtly they also break the symmetry because each parameter mentioned in rule 2 derives its assurances from only the specific corresponding parameter. We can restore the symmetry to the definitions by changing the definition of subtype, conforms, and method call assurance to read:

1. An interface \( J \) is a subtype of an interface \( I \) iff each of the signatures in \( I \) conform to at least one signature in \( J \). The requirement for nominal declaration is removed. In the reformulation of the conformance rule to, next, it would be difficult to accommodate nominal subtyping.

2. A signature \( m(I_0, I_1, \ldots, I_n) \) conforms to a signature \( h(J_0, J_1, \ldots, J_k) \) iff \( m=h \), \( n=k \), and for a \( 0 \leq i \leq n \) each signature in \( I_i \) conforms to a signature in some \( J_i \). This allows the assurances indicated by the signatures to come from any of the parameters rather than simply from the expected (same) ones. We call the phenomenon occurring when an assurance in an \( I_i \) conforms only to one in a different \( J_i \) recombinance.

3. A method call, written \( m(p_0, p_1, \ldots, p_n) \) is assured if for some \( 0 \leq i \leq n \) \( p_i \) is not null and \( m(p_0, p_1, \ldots, p_n) \) is assured by the interface \( P_0 \) declared for a variable \( p_i \).

Like the assurance of a method call, recombinance can occur only for assurances associated with parameters that may not be null references. Such parameters are indicated to be necessary, otherwise they are supplementary. To promote recombinance, the language allows and encourages declarations in which references that must not be null are declared necessary. When interpreting interface declarations from the common model, the target

---

1. It plays approximately the role of the arrow object of the method in [13].
The parameter is clearly necessary, but other reference parameters must be assumed to be supplementary unless additional information is provided.

The effect of these definitions is far reaching and can be surprising. For example, the two interfaces \( A = \{ f(A,B) \} \) and \( B = \{ f(B,A) \} \) are equal. Sound semantic foundations for this sort of type system were laid in [12] and algorithms for dealing with them have been explored recently in [11].

The insightful reader may worry at this point that eliminating method-parameter asymmetry in this way may cause too much ambiguity. In fact, without making additional distinctions we would witness parameter-collapse – all same-named methods with the same number of parameters and the same set of requires assurances become the same method. In later sections of this paper, we will introduce other language features that help reduce the ambiguity, but do so in more intention-oriented ways than did breaking method-parameter asymmetry.

### 4.1.3 An Example

Although other concepts will be introduced later, it would be well at this point to try to show how the concepts introduced thus far work in the context of an example to which we can add later. To motivate the example, we present a scenario where a small business, called “SoleTrader” is operating but is subsequently taken over by a larger entity, called “Superstore”. Some of the software in operation in the acquired business needs to be retained, but some of the supporting function needs to be replaced by the larger business’ implementations. The software systems employed by both businesses are independently developed and, although similar in general, they differ in some of the implementation choices. While the core functions of both businesses are effectively identical, the manner in which they conduct operations and organize themselves may be slightly different.

### SoleTrader Interfaces

```java
interface Empty{}
interface Ordering { 
  reorder (Empty store);
}
interface Sales { 
  sell (Ordering item, Empty customer);
}
```

The interface definitions above form a very simplified view of the sole trader’s system. It defines three interfaces: `Ordering`, `Sales`, and `Empty`. The `Ordering` interface indicates the assurance of being able to `reorder`. The `reorder` method takes two parameters, `item` and `store`. Declaring that `item` supports `Ordering` indicates that the methods in `Ordering` can be called if `item` is non-null. The second (first non-target) parameter references the particular store reordered for. The fact that `store` supports the `Empty` interface indicates that it brings no assurances with it. In terms of our formulation above, `Ordering=\{\text{reorder}(\text{store})\}` and `B=\{\text{reorder}(\text{store})\}` are equal.

### Superstore Interfaces

```java
interface Empty{}
interface SuperOrdering { 
  reorder (Empty store);
}
interface TheBusiness { 
  reorder (Empty store);
  sell (Empty item, Empty customer);
}
```

Next we present the interfaces characterizing Superstore’s system. There are two main differences between the way Superstore’s consultants and SoleTrader’s consultants defined their systems, arising because the superstore has a different view on how a business is conducted. The first difference is that Superstore employs an interface for `TheBusiness`. This combines all the methods needed by the superstore. The second difference appears in the signature of `sell`, where the `store` reference is expected to bring with it all the assurances of `TheBusiness`, so no other parameters need bring any. So, `TheBusiness={\text{reorder}(\text{store})}`, `sell(\text{store},\text{Ordering},\{\})`.

Using our symmetric definitions for assurance, subtyping and conformance, the client code remains operational, regardless of whether the sole trader or the superstore service definitions are being used for the servicer. After a takeover, the superstore can continue to use the exact same client code for the same operations because of the way it has defined its interfaces. The combination of a symmetric definition of interfaces and the use of a symmetric call simplifies the integration of systems.

For the client, we still have the definitions from before. The new implementation still supports the calls because the client’s `sell(\text{store},\text{Ordering},\{\})` conforms to its `sell(\text{store},\{\})` and the client’s `\text{reorder}(\text{store})` conforms to its `\text{reorder}(\text{store})`.

Although we don’t show the case in detail, client code written using Superstore’s interfaces would also still be correctly supported by the servicers written by SoleTrader. The Superstore client would be using a call with the parameter types in `sell(\{\text{reorder}(\text{store})\},sell(\text{store},\text{Ordering},\{\}))`, which would conform to the SoleTrader’s `sell(\text{store},\text{Ordering},\{\})`.

### 4.1.4 Restoring Key-reference Symmetry

The second important symmetry broken by the common model for object-oriented is key-reference symmetry. Key-reference symmetry requires that the ability to associate assurances with references be independent from the representation of the reference as a pointer. It is a fact that implementations of methods in objects ultimately need to use references represented as pointers in order to address their fields. But this fact should not force the resolution code back into the clients. Consider the invocation of the `sell` method in the client example above. It is unlikely that the

```java
SoleTrader Client Code
Ordering item = ... initialization ...
Sales store = ... initialization ...
Empty customer = ... initialization ...
store.sell (item, customer);
item.reorder (store);
```
physical item purchased or the customer actually came equipped with a pointer to the object(s) representing them in the client or server. It more likely came equipped as a bar-code that can be read as a string. In the common model for object-oriented software, the target for the method call must be converted from the bar-code to pointer-based reference. Even with the symmetric model discussed above, if only pointer-based references are allowed to carry assurances then at least one of these externally-provided values will have to be converted to a pointer, and the question arises … which one? We could find that the client has any one of these code sequences embodied in it:

```
1. item = ItemRegistry.find(itemBarCode);
   item.sell(storeCode, customerId);
2. store = StoreRegistry.find(storeCode);
   store.sell(itemBarCode, customerId);
3. customer = customerRegistry.find(customerId);
   customer.sell(itemBarCode, storeCode);
```

In all cases, the decision about which key must be converted into a reference became part of the code in the program’s logic, perhaps no longer even local to the method call, and certainly difficult to change. This fact indicates that to restore the malleability inherent in the algorithm’s essential logic, we must avoid having the language force breaking the symmetry of choice. We can restore key-reference symmetry by allowing computational data like strings representing bar-codes to be used as references, displacing their resolution until the method call is dispatched to a service. We call these generalized references.

There is a second important reason for introducing generalized references. When features or aspects are composed to allow several different services to provide behavior, the reference may refer to more than one object at any time. The choice of which one to use cannot be made until after the dispatch. For example, if the call must be dispatched to both to a purchaser’s attorney and a seller’s attorney, they may each use a different object to represent the same party. Embedding the choice in the client who does not even know the structure of the services greatly reduces the flexibility of the client software.

4.1.5 Mints

All around us we find examples of references – people are given identifying codes by governments, employers, and even businesses. Objects of commerce are given bar-codes. Shipped items are assigned shipping numbers, purchase orders get order numbers, etc., and even abstract computer representations like “user”, “device”, “message”, or “RemoteDataObject” are assigned references when they are created. Many of the values that are used to refer to the items in the real world do not refer to unique computer representations, and even the references to abstract representations often must address multiple “aspects”.

Mints provide a set of values, unique within their purview. They implement one of the two functions of conventional object-creation constructs like “new” – producing identification. The resource allocating function of “new” is separated and will be discussed later. Mints are identified by URL’s, though syntactic sugars are applied. Although not part of the language specification, some mints are designed to allow for proxies, some of which may be process-local for performance reasons. Interaction between the language interpreter and the mint allows mints to use either garbage collection or explicit allocation/deallocation strategies, and they may control the representation of references for this purpose or other purposes like security. Some mints are built-in to the language, as is the conventional local-store-based pointer reference mint that allows for garbage collection.

Mints may allow for a variety of representations to be used to hold the values they mint, but the representation chosen for a reference is not part of an outerface declaration or involved in type-matching. It must, however, be declared as part of any variable declaration: parameter variable, local variable, or field, so that appropriate resources can be allocated. Representations for references need not be managed by the mint itself, although it may participate in determining their suitability.

4.1.6 Type Model, Parameter Declarations, Outerfaces

The mint specification for a reference is part of the declaration of every variable or method parameter. However, the representation of the reference is not part of a method’s signature in interface specifications. In reinterpreting an existing OO language, like Java or C++, all references are interpreted as arising from the built-in local-storage mint. But to provide key-reference symmetry other mint declarations should be used when they are known. Declarations using other mints also reduce the problem of parameter collapse because they add additional information to signatures.

The extension of interfaces that provides the full expressive power of our type model is syntactically distinguished by the keyword “Outerface”. An outerface specifies a set of assurances, each of which specifies a method name and parameter types. Each parameter type is either a primitive type or a reference type. A reference type declaration specifies a mint, a necessity, and an outerface, although the mint and necessity can be defaulted to the local-memory mint and “necessary”. All parameters to the assurances are written explicitly, without insertion of the implicit target-interface described for interfaces in Section 4.1.2. An outerface has the syntactic structure as:

```
Outerface outerfaceName [variableName, ...];
```

Let us rework the example, with slight variation, using outerfaces.

4.1.7 Reworking the Example

We explain the use and value of outerfaces by returning to the store example to demonstrate how two entities, with alternative organisational techniques, may use the same client code seamlessly.

```
SoleTrader Outerfaces

Outerface Empty{};
Outerface Ordering {
   reorder (Ordering Barcode item,
            Empty StoreCode store);
}
Outerface Sales {
   sell (Sales Storecode store,
        Empty Barcode item,
        Empty LoyaltyID customer);
}
```
The definition above is almost the same as the earlier definition of the sole traders system given in terms of interfaces, but indicating varied mints for the references. As before, the sole trader defines three outerfaces; Ordering, Sales and Empty. The Ordering outerface signifies the assurance of being able to reorder. This method takes two parameters; item and store. The item is a reference that was originally issued by the Barcode mint. Declaring a variable that supports Ordering indicates that the methods of the outerface can be called. The second parameter is another identifying that references the particular store that was originally issued by the specific Storecode mint. The fact that it supports the Empty outerface indicates that it brings no assurances with it.

The Superstore’s outerface is then almost the same as earlier.

```plaintext
Superstore Outfaces
Outerface Empty{};
Outerface SuperOrdering {
    reorder (Ordering Barcode item, Empty Storecode store);
}
Outerface SuperSales {
    sell (TheBusiness Storecode store, Empty Barcode item, Empty LoyaltyID customer);
}
Outerface TheBusiness {
    reorder (Ordering Barcode item, Empty Storecode store),
    sell (TheBusiness Storecode store, Empty Barcode item, Empty LoyaltyID customer);
}
```

The client code shown here has state variable declarations for local variables holding references represented as Strings.

```plaintext
SoleTrader Client Code
Ordering Barcode String item = ... initialization... ;
Sales Storecode String store = ... initialization ...;
Empty LoyaltyID String customer = ... initialization... ;
```

As above, after a takeover, the superstore can continue to use the exact same client code for the same operations because of the way it has defined its outerfaces. The argument justifying that for reorder is essentially the same as before, but the argument for sell is subtly different because the first argument to Superstore’s sell is TheBusiness instead of Koa. So the conformance demonstration for sell must demonstrate that reorder is also assured. Fortunately, it is present in SoleTrader’s Order interface, defined for item.

The call to the sell method will succeed in this case because the store reference brings with it the sell assurance as defined in the sole trader’s Sales outerface. This means an appropriate method can be found. It should be noted that the item reference also provides the ability to call reorder. In Continent, references not only denote an object, but also a set of assurances about methods that can be called. These assurances are simple facts and are themselves independent of the original reference if it is not null.

### 4.2 The Use of Context in Dispatch

The issue raised in Section 3.1 arises from noticing that many emerging areas of software imply that messages requesting service are delivered to servicers by dispatch mechanisms able to apply more intelligence than simply delivering the message to a “target” object. We envision an overall system structure like that shown in Figure 1.

![Figure 1 - Federated Service Dispatch](image)

Figure 1 - Federated Service Dispatch

Services are the fundamental units with which the dispatchers deal. Services provide bounds for defining tolerable vs. intolerable ambiguity. A single method call may be routed to many services simultaneously, in sequence, or in some orchestrated manner controlled by the dispatch rules. This is not considered to be an undesirable ambiguity in service choice, but to be instead the mechanism for providing orchestration of multiple activities that must take place in response to a method call. Any dispatcher is only one part of a federated collection of dispatchers. These dispatchers communicate with each other to pass on information about clients’ needs and servicers’ capabilities.

Servicers function within a context controlled by a dispatcher whose rules are determined outside the language of any of the clients or servicers. The dispatch rules might derive from AspectJ-like “advice” passed to the dispatch federation when the service containing the advice starts. The rules might derive from dynamic negotiations with a service registry. The dispatchers themselves may employ any necessary context to select servicers and/or other dispatchers through which the messages must pass, and they must communicate enough information to allow for optimized execution paths competitive with conventional dispatch.

We can no longer imagine that all services are programmed in a single language or even within a single paradigm. However, it is necessary to specify the meaning of the Continuum language in closed form, describing how an implementation within a single service is resolved.

### 4.3 The Servicer’s View

#### 4.3.1 Classes and Modules

As the discussion of the client view began by analogy to the interfaces used in the common view, the discussion of implementations is best begun by analogy with the classes used in the common model. A class defines state variables and method implementations that use the state variables. Definitions of
method implementations are declared with a name and list of parameter variable declarations. Modules are the syntactic extension of classes that allows expression of the full semantic capabilities of Continuum. The method implementations in classes have an implicit “this” parameter, while those of modules do not.

Parameter variable declarations are like state variable declarations shown in the sample client code in Section 4.1. Each parameter variable is declared to have a primitive type or a reference type. Reference types are typed with a mint, outerface, and necessity, and are also given a representation. Class or Module names may be used anywhere that interface names may be used, but they are interpreted to refer to the minimal supported interface that is inferred from the class definition. In the case of a class, the type of the target parameter (this) is specified using the class name.

The use of a class’s name to qualify one of its methods’ parameters indicates the unique phenomenon called decapsulation. – the opening up of the private representation of an object on entry to one of its methods because the dispatch process has insured that the parameter is, in fact, the right class. Decapsulation occurs when a parameter to a method implementation is declared using the name of the class or module containing it. It can be prevented by prefixing the class name with the keyword “Outerface”. If decapsulation is indicated, the parameter must have a local-memory pointer representation and it is permissible to use that parameter to address state variables declared in the class or module. The fact that decapsulation can be applied only to references indicating the implementing class has a significant implication: access to state variables through references to other classes must be accomplished using get/set methods.

4.3.2 Inheritance and Classification
The term “inheritance” is used indiscriminately for interface inclusion, class implementation derivation, and dispatch specialization. Because Continuum uses a structural type system for interfaces, the concept of interface inheritance is defined as subtyping, as described above. No separate declarations are needed. Determination of whether a class implements an interface is defined similarly, without explicit declaration. Continuum separates the concepts of derivation and categorization for purposes of creating classes and for ambiguity resolution in message delivery. Derivation is concerned with the reuse of implementations, allowing the state and logic in an implementation to be reused from other implementations. Specialization determines to what category of thing this belongs for purposes of determining which of several possible implementations within the service is chosen in preference to others, or whether an ambiguity exists.

4.3.2.1 Derivation and Overriding
The dispatch specifications form only a part of the larger language for specifying how service implementations are to be reached from service calls. We use classes to compatibly specify simple modules, providing a simply specified module structure for prosaic implementation and dispatch specification. A class “Exx” can be is declared with the syntax:

```
Class Exx extends Object {
  new fields and method implementations
}
```

The semantics of “extends” are familiar: Exx contains all of the implementation elements from Object, which are called a “superclass” of Exx.

If a superclass defines a method and a subclass defines a method to which it conforms, a static ambiguity is diagnosed. (This situation can only arise in modules because the target parameter is assigned the class type in the case of classes.) The diagnosis can be alleviated if the subclass employs the keyword “overrides” with respect to the superclass’ method. The override keyword specifies an outerface (a set of signatures) that the subclass’ method is intended to override. This set implicitly contains the subclass’ method, which does not need to be specified explicitly. The class overrides (excludes the implementation from a superclass) methods that are listed in the overrides clause.

Modules can extend several superclasses. If a method that could be derived from a superclass conforms to another method that could be derived from a different superclass, a static ambiguity is diagnosed in the subclass unless both are overridden in the subclass.

Within the same class, it is legal to provide two methods where one but only one of them conforms to the other.

4.3.2.2 Specialization.
Specialization is a declared relationship between classes, but it induces a relationship between methods. Method M is said to specialize method N if N conforms to M and the class containing M specializes the class containing N. If a method call could be validly dispatched to either of two methods, it is dispatched to the more specialized. This provides the expected behavior that a method in a subtype will be executed in preference to the same method in a supertype. Class specialization is a transitive relationship that is not allowed to be cyclic.

```
Module Exx extends OneObject, Another
  specializes AThird, AFourth {
    new implementations
  }
```

Assume that a method call resolves statically to the signature S, but that the evaluation of the parameters resolves to signature D. The dispatch candidates are the set of methods conforming to S that does not contain any method that is specialized by another conforming to S. The presence of at least one candidate is assured by the assurances checked by static type-checking. If the set has more than one element, the message would be ambiguous. The static type checking for continuum is intended to provide controlled ambiguity—allowing a message to be delivered to several implementations in different services, but preventing a message’s delivery to more than one implementation within a service. This is discussed further in Section 4.4.

A class is a module defined using the same shorthand for eliding the “target” that was as described for “interface” in Section 4.1.2. As part of a simplifying shorthand, its specialize specifications are taken to be the same as its extends specifications.

4.4 Services
Services wrap a collection of modules together into a coherent whole. A service is defined by its base modules used, exports, requires, resource specification, and ambiguity resolution.
4.4.1 Base Modules Used
The service defines a list of class, module, interface, or outerface names associated with the service. Syntaxic sugars providing for wild-cards, supertype closure, superclass closure, etc. are included. Modules are not “contained” in a service. Although they are associated with a service, they may be associated with more than one service. They simply provide a definition for collections of state and behavior.

4.4.2 Provides and Requires
The provides and requires outerfaces are provided by listing defined interfaces or outerfaces, using the same syntax as used for specifying the base modules.

4.4.3 Resource Specification
Some of the methods within the associated modules may require decapsulation of some of their parameter references, as described in Section 4.3.1. The specification for performing decapsulation is specified as a method call. This allows it to be implemented using whatever algorithm and state are appropriate to the task, whether database lookup or allocation or reuse of local memory. It converts whatever representation is used for the reference into a pointer-based reference to state information stored within the service. Services may provide different conversions for references from the same mint, associating them variously with different modules.

4.4.4 Ambiguity Resolution
Two forms of ambiguity can arise in defining a service: internal and external. Internal ambiguity denotes potential ambiguity arising from method calls within the service. External ambiguity denotes potential ambiguity arising from calls outside the service that are made to exported methods. Internal ambiguity is diagnosed on the basis of the whole service – all the calls and definitions are known. Much work has been done on ambiguity with symmetric multimethods, discussed in Section 7.2. The calls giving rise to external ambiguity can not be seen and form an open-ended set, so they must be presumed to arise. Diagnostics indicate which ambiguities must be resolved. Potential ambiguities can be resolved by specifications within the resolution part of the service that add specialization relationships and, if necessary provide methods that cover the otherwise ambiguous cases.

4.4.5 A Service Example
We can illustrate the service-implementation concepts by describing implementations for the sole trader’s sell and reorder methods. Some of the implementation here is not intended to be sensible, but just to help demonstrate concepts. The classes in SoleTrader’s implementation of the services resemble its outerfaces in Section 4.1.7. The implementation bodies are omitted for clarity. The services uses two classes: Restocker and Register, shown here. In addition, it requires the support of described by the SystemLibrary.* outerfaces. The minimal supported outerface of the Restocker class is identical to the Ordering outerface, even though it is not mentioned explicitly. It supports Ordering even without an explicitly declaration. Even though not sensibly called for, the minimal supported outerface of Register includes an implementation of reorder in addition to Restocker’s implementation. If the service exported no outerfaces, the fact that it contains no calls to reorder would avoid diagnosing an ambiguity. But reorder is contained in an exported outerface. (In fact it is contained in two, but the effect is the same.) In the absence of remedy, this would give rise to a static ambiguity – calls from outside would see two possible implementations. The indicated remedy is that Register specializes Restocker. That fact need not be declared in the class definition. Unless a pair of classes is intended to appear together generally, for example if one class extends another, declaring that one specializes another has the disadvantage of making them inseparable for other uses. The classes in this service are described using the “interface/class” shorthand, and so there is only one mint involved – the built-in local storage mint. Hence, the decapsulation of the “this” parameter to the methods requires no resources. A specification of resources is shown for completeness.

<table>
<thead>
<tr>
<th>SoleTrader Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Restocker {</td>
</tr>
<tr>
<td>int count;</td>
</tr>
<tr>
<td>reorder (Empty StoreCode store) { ... }</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Class Register {</td>
</tr>
<tr>
<td>sell (Ordering item,</td>
</tr>
<tr>
<td>Empty customer) { ... }</td>
</tr>
<tr>
<td>reorder (Empty StoreCode store) { ... }</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Service SoleTraderImplementation</td>
</tr>
<tr>
<td>Uses Restocker, Register;</td>
</tr>
<tr>
<td>Provides Restocker, Register;</td>
</tr>
<tr>
<td>Requires SystemLibrary.*;</td>
</tr>
<tr>
<td>Resources;</td>
</tr>
<tr>
<td>Resolutions Register specializes Restocker;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
To illustrate resourcing, let us look at this implementation as it might have been done natively in Continuum for Superstore. Other than indicating the mints for parameters, the class definitions look similar. We used “extends” rather than providing reorder explicitly in Register. In the absence of an explicit “specializes” clause, the “specializes” is taken to be the same as the “extends”, making the separate resolution unnecessary. Note that the method implementations specify their class on some of their parameters. This causes the parameters to be decapsulated, giving, for example, reorder’s implementation access to count through the item parameter.

<table>
<thead>
<tr>
<th>Superstore Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Restocker {</td>
</tr>
<tr>
<td>int count;</td>
</tr>
<tr>
<td>reorder (Restocker Barcode item,</td>
</tr>
<tr>
<td>Empty StoreCode store) { ... }</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Module Register extends Restocker {</td>
</tr>
<tr>
<td>sell (Register Storecode store,</td>
</tr>
<tr>
<td>Empty Barcode item,</td>
</tr>
<tr>
<td>Empty LoyaltyID customer) { ... }</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Service SoleTraderImplementation {</td>
</tr>
<tr>
<td>Uses Restocker, Register;</td>
</tr>
<tr>
<td>Exports Restocker, Register;</td>
</tr>
<tr>
<td>Requires SystemLibrary.*;</td>
</tr>
<tr>
<td>Resources Pointer Barcode(String) { ... };</td>
</tr>
<tr>
<td>Pointer Storecode(String) { ... };</td>
</tr>
<tr>
<td>Resolutions;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

5. MOVING ONWARD COMPATIBLY
Continuum aims to provide a software model that is more in-line with a number of needs for change from the common model used
for object-oriented software that are emerging. But we have deliberately structured its semantics in a way that can provide a proper reinterpretation of common-model languages like Java or C#, enabling us to conserve the investment in existing software and training. Three levels of translation are expected to be useful: a minimal reinterpretation of existing languages to permit the reuse of legacy code, an augmented reinterpretation that allows compilation of programs in a common object-oriented language with additional deployment information that changes its semantics in some desired way, and a syntactic extension of an existing language to allow expression of the full power of the reinterpretation. Comparison with Java is used for example.

5.1 Strictly Extended Reinterpretation
Consider a strictly extended reinterpretation of Java. As alluded to in earlier sections, to maximize compatibility several defaults are applied:
1. the target parameter of methods is treated as necessary while all other reference parameters are treated as supplementary.
2. extends is taken to imply specializes for class definitions.
3. all references are defaulted to the local-storage mint
4. get/set methods are introduced sub-rosa, and employed when it is necessary to access state information of references declared using classes rather than interfaces

The primary incompatibilities that remain with this translation arise from introspective practices, whether by use of Java’s reflection interfaces or by language features such as the “instanceof” predicate. Access to the reflection capabilities can continue to be supported by producing appropriately compatible meta-information and assuring that it is possible to call Java-compiled class files from Continuum-produced class files through a compatible linkage. This is facilitated by the fact that most of the linguistic differences are in type-checking rather than in runtime representation.

The objective of the strictly extended reinterpretation is that correctly-functioning Java programs continue to function as before, interoperating with other programs in the more relaxed manner described by the client model described in Section 4.1. The extended reinterpretation is deliberately intended to be incompatible in the other direction, however. Some programs that were not correct Java programs will also function in this way. The change from “Sole Supplier” to “Superstore” described in Section 4.1.3 is an example of this kind of difference.

5.2 Augmented Reinterpretation
Augmented reinterpretation is the compilation/interpretation of a Java-expressed program with side-directives that relax one or more of the list of defaults described above in Section 5.1. The primary expected use is to treat some reference parameters other than the target as necessary. This would permit increased flexibility through recombinance. Other uses may be to exploit multiple “extends” for classes (while retaining a single “specializes”). This would permit better reuse of “mix-in” classes without disturbing Java’s single-inheritance view of dispatch.

5.3 Continuum
Continuum compilation allows extended use of the Outerface, Module, Mint, and Service constructs, and interprets Java as a syntactic subset in which all reference parameters default to necessary and separate multiple extension and specialization are possible. Reflective Java constructs are not supported in Continuum, because they do not reflect the underlying semantics of Continuum.

6. SECONDARY FEATURES
We foresee adding other features to the language that may not be immediately required to address the needs of the emerging variations, but can be seen as secondary because they address improvements that are useful or consistent with one or more of these needs.

6.1 Method Description Glossaries
One of the uses given to interfaces and classes in the common model is to help distinguish otherwise identical methods. Our use of structural typing rather than nominal typing has made this use unavailable. But its use for this purpose in common object-oriented languages was never sound — generally multiple-inheritance of interfaces or implementations coalesced meanings that were otherwise thought to be different. To help assure that method callers and implementations are talking about the “same method” in Continuum and to enhance the specification of system structure, Continuum allows each message’s name and type signature to be augmented with several specifications: results, requirements, suggestions, characteristics and conflicts. Each of these is a set of references to terms in a separately specified glossary. Local references to methods must refer unambiguously to a static declaration of the method, at which point the additional information is incorporated.

The glossary provides a shared definition of terms which couples a symbolic reference that can be employed efficiently in method specifications with a natural or formal language statement that developers can use to decide upon similarity or difference.

In fact, the concept of “same method” is a subtle and deeply troubling one even with such a simple extension to object-oriented as is provided by aspect-oriented software. To the caller of a method, the important function provided by a call may be narrow, something like “records the book in the library catalog”. But to the designer of a system, the interesting characteristic may be “is one of those points at which a document should be published”. Several coordinated implementations are woven together to achieve several coordinated requirements. Aspect-oriented software technology has tried to get at this issue in several ways. This notion of coordinated, woven behaviors was fundamental for subject-oriented programming [7], but the use of simple names rather than intentional specifications was not available in the language-independent context it addressed. Aspect-oriented programming [10] can use patterns applied to the code to provide the equivalent of these intentions. Continuum offers the possibility of recording them directly when they are known in advance and using them to find and combine appropriate services.

6.2 Method Hierarchies
In the common object-oriented languages, interfaces are arranged in hierarchies, but methods are treated as simply different or identical. In the service-oriented approach, the messages themselves unavoidably form hierarchies based on their assurances. We use that fact to extend their power. So if outerface “B” is a subtype of outerface “A”, then message “m(B)” is a
specialization of message “m(A)”. But Continuum allows specializations to be explicitly declared as well, indicating, for example that message “m(A)” is a specialization of message “m()”, or even of message “m(n(A)”.  

7. RELATED WORK
The work described here is clearly related to the elements emerging variations discussed in Section 2: to aspect-orient programming,, ubiquitous/pervasive computing, service-oriented software, grid computing and event-driven software. They all share a conception of groups of objects communicating with one-another through a message-dispatching structure capable of taking contextual issues into account. The work here shares this conception and also shares with several of them the use of a packaging structure larger than a single class, whether called aspect, service, locale, or something else. But to this shared body of concepts, contributions are added from areas with which the solutions adopted here have similarities to other work. These related areas have to do with: multimethods, recursive structural types, connectors and resource provisioning,. We address these related areas in turn.

7.1 The Shared Base
This work is distinguished from the shared base technologies in that it removes all code related to several infrastructural issues from the “application logic” written in methods. The issues primarily have to do with symmetry-breaking. The advantage to this removal is threefold: 1) the application developers do not need to become aware of concepts like containers, connectors, or proxies involved in working with those issues, 2) the code itself does not embed choices that inhibit its reuse in different circumstances, and 3) the logic of the methods is clearer and more understandable to later maintenance developers.

7.2 Multimethods
One of the infrastructural issues removed from the clients is dependence on how the method implementation will be determined. This phenomenon is generally termed generic call [9], or, when the dispatch may use information from several parameters, “symmetric multimethods” [5]. Extensive investigation of symmetric multimethods has been made in connection with a family of programming languages like Cecil [4] and Dubious [13], which indicates, among other things, that in the presence of symmetric multimethods, it is either the case that diagnosis of ambiguity must wait until all code for a system is seen or it is the case that implementation methods can not be allowed to cover the space of possible call situations in an arbitrary manner. Several rules limiting the covering are shown to prevent ambiguity. The two remaining alternatives are to tolerate undiagnosed ambiguity, as do CLOS [9], or disallow any potential ambiguity, as does MultiJava [3]. Continuum provides a controlled combination of these alternatives, although differently than does and Relaxed MultiJava [14]. Continuum differs most significantly from these languages in its use of recombinance of the assurances from different parameters to introduce flexibility. Allowing one of the parameters to supply an assurance for a method that the client had “expected” to come from a different parameter substantially enhances the modularity of the software written in Continuum. Recombinance not only allows less-restricted method call, but allows the assurances to be shuffled among the parameter variables.

Continuum also differs from these languages in using the “service” construct to diagnose ambiguity on a scale in-between that of class/module and “whole system”. Services were introduced to address resource provisioning issues, but they provide a way to “divide-and-conquer” the type-checking problem. Each service must offer no potential ambiguity when viewed from outside. But internally, only ambiguity owing to the actual usages needs to be repaired. A simple mechanism is provided for defining the repair without actually changing the classes involved.

Continuum also differs from most of these languages (with the exception of the MultiJava family) in that it adopts the object-oriented need for what might be termed “conditional safety,” rather than the “unconditional safety” of the Cecil family. Conditional safety does not requires that an implementation can always be found. It requires only that an implementation can be found if the qualified reference is not null. Qualified safety allows clients to be executed in circumstances where implementations for some interfaces on which the client might depend are missing but where that circumstance is acceptable because no instances are created that call for it.

7.3 Recursive structural types
Continuum employs recursive structural typing for the outerfaces that characterize references. Although nominal typing has been the mainstay for object-oriented programming, its use significantly reduces the malleability of the software produced. However recursive structural types have been heavily investigated in the past. Their soundness has been demonstrated [12], and algorithms for efficiently dealing with issues of comparison and subtyping have been thoroughly explored [8,11]. That work has been applied to data typing, but we apply it here to method signature typing in interfaces, which is an equivalent problem.

7.4 Connectors and Resource provisioning
Possibly because it has generally been thought of as applying uniquely to local storage, the issue of supplying resources to back references to objects’ state has generally been invisible to programming languages, kept in the infrastructure. But networks of business services and the rise of dynamic and mobile network configurations have begun to burden the application software with the management of resolving data keys to the appropriate objects, and of managing the proxies that “hide” the indirects. For comparable technology we can look at Enterprise Java, in which we find the J2EE’s “Collection” [20] which must be managed in client application code to find appropriate proxies or the Java Remote Method Call Technology [19] which creates so much complexity in client code for proxy management that tools [1] have been developed to help automate the code generation. Significant problems arise in trying to extend garbage-collection to generalized references. The issue is generally finessed today except with respect to what we term the local-storage mint. At worst, we will not improve that situation. At best, we offer a setting for further investigation of multiple differently-managed heaps [2,15].

There are strong dualities between the object/method and the port/message views, so that the dual of proxies can be abstracted as connectors. With this related this view of software, tools have also been developed to help automatically produce and match connectors [1]. But the port model suffers from the same fragility.
as does the object model – the client software must know which ports lead to desired services, and must take the burden of using data keys to find or create the appropriate ports. The fact that service client software solidifies decisions about the choice of which port to use and which data to use to find them produces the same loss of symmetry we described in Section 4.1.1.

8. SUMMARY AND FUTURE WORK
The System Structure Group at Trinity College Dublin is exploring programming languages and technology supporting service-oriented software development. It is at an early stage in an overall agenda that includes defining, implementing and evaluating solutions for language issues like those presented here. Continuum contains a number language features, like its use of assurances and recombinance and its provision for generalized references, that promise to make it possible to create software which is significantly more malleable than previously practical, without making significant performance or optimization compromises beyond those already made for object-oriented programming in general.

We consider it to be extremely important for the eventual acceptance of any new language features that they can be introduced without losing existing investments in software or in developer training. The tactic presented here of making a fundamental change that can be introduced by reinterpreting existing programs in a compatible but forward-extensible manner appears to offer great promise.

The group is developing a virtual machine for Continuum, consistent with the exploitation of Java as a subset. Language features will be added incrementally, eventually including resolution of other symmetries broken in most current languages, as does the object model – the client software must know which ports lead to desired services, and must take the burden of using data keys to find or create the appropriate ports. The fact that service client software solidifies decisions about the choice of which port to use and which data to use to find them produces the same loss of symmetry we described in Section 4.1.1.

9. REFERENCES
12. Lewis C. and Rosen B. K., Recursively Defined Data Types. In Conference Record of (First) ACM Symposium on Principles of Programming Languages, Boston, October, 1973, pp. 125-138