# Interaction Refinement in Object-Oriented Systems
# (Extended Abstract)

Neelam Soundarajan

Computer and Information Science
The Ohio State University
2015 Neil Avenue Mall
Columbus, OH 43210
USA

e-mail: neelam@cis.ohio-state.edu
Tel: (614) 292 1444. FAX: (614) 292 2911

**Abstract:**

An OO designer typically starts with a high-level idea of the interactions between the key objects in the system. As the design progresses, the designer *refines* these interactions by identifying the exact operations of each object that other objects will invoke and the order of invocations, or by introducing new objects with appropriate operations to mediate the required interactions between existing objects, etc. These refinements are usually not recorded, so the rationale behind the design maybe lost. We motivate the notion of *interaction refinement* with a few examples, provide a precise definition of the concept, and develop a formalism that can be an important tool in recording and validating OO designs.

# 1 Introduction and Motivation

'*Stepwise refinement*' which we will call *procedural refinement*, is one of the most important tools in procedural design. We believe that *interaction refinement* plays an equally important role in OO design. To see what we mean by interaction refinement consider a simple example. Suppose we were designing a system consisting of a 'client object' *CO* and a 'server object' *SO*. The high-level specification of the system might say that the purpose of the first interaction between these two objects is to identify the client to the server. The system designer may decide to map this interaction into a series of interactions between *CO* and *SO*. First *CO* might be required to supply its 'user-id', and then its password; if *CO* provides an invalid password, *SO* might allow *CO* several chances to get the password right; etc. The designer may also choose to introduce new objects to mediate this interaction. For instance, the authentication of the password may be done by an authentication object *AO* that encapsulates this task. Indeed it might be a better design to have *CO* send the password directly to *AO* rather than via *SO*, since the protocol on how many attempts the client is allowed to get the password correct etc., can then be encapsulated in *AO* which is probably where it belongs. In any case, what we have done here is to refine the high-level interaction *identify-client* between *CO* and *SO* into a series of interactions involving *CO*, *SO*, and *AO*. This is an example of an interaction-refinement-step. In the rest of the paper we will often abbreviate interaction refinement to *IR*.

There is another type of refinement which we will call *structure refinement*, or *SR* for short, which is quite different from *IR*. In a step of *SR*, we design a particular object in the system, say *SO* of the last paragraph, to consist of a number of (member) components and implement the operations of the object in terms of the operations on its components. In this type of refinement we are not refining the external interface of the object in question; rather, we are refining its *internal structure*. Almost by definition, this refinement has no effect on *CO* and other objects external to *SO*. By contrast, in a step of *IR*, we refine a high-level interaction between two (or more) objects of the system into a series of low(er)-level interactions between these and possibly other objects in such a way that these low-level interactions achieve the intended purpose of the high-level interaction.

The key question is, how do we know that the 'intended purpose' of the high-level interaction is indeed achieved by the sequence of low-level interactions that we have refined it into? In other words, what does it mean for a particular interaction refinement to be correct? We will address this question in the next section after first introducing the notation that we will use for specifying interactions, and defining precisely the notion of interaction refinement. We then present a rule that will allow us to formally show the correctness of a given refinement. In the case of *SR*, the corresponding question would be, how do we know that a particular refinement of the structure of a given object into certain specific component objects and the implementation of its operations in terms of operations on these components, results in the object exhibiting correct external behavior? This question has been addressed by many authors [3, 5, 6] and several formal systems proposed to establish the correctness of a structure-refinement-step.[1]

[1]The corresponding question for the case of *procedural refinement* was answered by the classic formalisms of Hoare, Dijkstra, and others.

It is worth noting at this point that the reason we prefer the term 'procedural refinement' (or *PR*) to the more standard 'stepwise refinement' is that what we do in a step of *PR* is to refine the procedure, or the sequence of actions, needed to perform a certain computation. Moreover, interaction refinement and structure refinement are also stepwise, i.e., in a complete system design there are usually several steps of *IR* and of *SR*. Thus the key difference between the three types of refinement is not that one is step-wise and the others are not, but in *what* is being refined. In the case of a step of *PR*, we refine a given high-level action or computation into a composition of lower-level computations. In the case of an *SR*-step, we refine the structure of an object into its component objects. In the case of an *IR*-step, we refine a high-level, i.e., abstractly specified, interaction between a group of objects into a sequence of lower-level interactions between these objects, and possibly other objects. All three types of refinement are important in OO system design.

Although the idea of interaction *refinement*, and the attempt to answer the question "what does it mean for such a refinement to be correct, and how we can we establish this correctness?", seem to be new to this paper, the importance of understanding the desired sequence of interactions between the various objects of a system, and letting this understanding guide the design of the system, is well recognized. Indeed, once the interaction sequences have been *fully* refined, what we have is essentially the complete collection of *use-cases* [4] for the system; and as Jacobson [4] argues, use-cases play an important role in designing the system. Thus the main contributions of this paper are to provide a notation for precisely recording the sequence of refinements that we go through in going from a set of high-level interaction sequences where the interactions between the objects are rather abstract, to arrive at the set of actual or concrete use-cases applicable to the system when it is finally built, and a precise method to establish the correctness of these refinements.[2]

The paper is organized as follows: In the next section we introduce the notation for specifying interaction sequences, define what an interaction refinement is, and present proof rules that will allow us to show that a given refinement is correct. In the third section we discuss (informally) two examples of OO systems and show how *IR* plays an important role in the design of such systems. In the final section we reiterate the importance of *IR* in OO design, and the need for a formalism such as ours to establish the correctness of the resulting designs. We also explain where *IR* fits in the standard software life cycles. Finally we note the particular importance of *IR* in distributed OO systems.

[2]It may be useful at this point to draw a distinction between what Jacobson calls 'abstract use-cases' and our notion of high-level interaction sequences. As we have explained, high-level sequences are what a system designer starts with. They consist of elements that represent a high-level of view of the interactions between the various objects in the system. Jacobson's abstract use-cases are not in any way related to these sequences. Abstract use-cases are instead portions of (final) use-cases that happen to appear in more than one use-case. Jacobson suggests that it would be useful to identify these so that we don't waste time implementing them more than once.

## 2 Specifying and Verifying Interaction Refinements

Consider an OO system $S$ consisting of a number of objects interacting with each other. We are interested in the various possible sequences of interactions between these objects.[3] Suppose $\lambda$ is a possible interaction sequence of $S$. Each element of $\lambda$ represents an *interaction* among the components of the system.

Two types of interactions, simple and compound, may appear in $\lambda$. A *simple interaction* corresponds to the invocation of an operation of one of the objects of $S$ by one of the other objects of $S$, or the return from such an invocation.[4] A *compound interaction* involves two or more objects of $S$ and does not correspond to a single call or return of an operation of one object by one of the other objects; rather it will be *refined* into a sequence of such calls and returns during one or more steps of *IR*. In the sequence $\lambda$, we will record a simple interaction that corresponds to the invocation of an operation of one object by another by specifying the name of the calling object, the name of the called object and the operation invoked, and any parameter values; a simple interaction corresponding to a return will similarly record any result returned to the caller. An element of $\lambda$ that corresponds to a compound interaction will be specified by providing an appropriate name (such as 'identify-client'), specifying the set of objects involved in the interaction (such as $\{CO, SO\}$), and the values of any parameters involved.

How do we specify the set of possible interaction sequences of the system $S$? We will use an assertion $I$ that is satisfied by *all* allowed interaction sequences of $S$ and *only* by these sequences. This latter requirement that only allowed sequences of $S$ satisfy $I$ is not critical to the development of our formalism. We decided to impose this requirement mainly in order to mirror the usual style in which use-cases are used [4]. We will return to this point shortly.

Suppose we have another specification $I'$ for $S$. We can now precisely state our basic question:

> What relation must hold between the specifications $I$ and $I'$ in order for us to be able to legitimately claim that $I'$ is a (interaction) refinement of $I$?

The obvious answer, that $I'$ is a refinement of $I$ only if it implies $I$ is wrong. There are two problems: First, as we just noted, if $\lambda$ satisfies $I$ then $S$ must actually be capable of exhibiting the behavior corresponding to $\lambda$. So $I'$ cannot be stronger than $I$ because then there might be a sequence $\lambda$ that satisfies $I$ but not $I'$, and $I'$ would no longer be a valid specification of $S$ since $S$ is capable of exhibiting the behavior corresponding to $\lambda$. This is clearly a minor problem, and if necessary we can solve it by reinterpreting a specification of $S$ to be something that is satisfied by all possible sequences of $S$ and omitting the requirement that only these sequences satisfy it. We won't pursue this point further in this paper (except to note how rule (1) below will be modified if we were to omit this requirement).

[3] Of course, in general objects of $S$ will also interact with agents –such as actual users– external to $S$; as is usual, we will represent these by introducing 'stand-in' objects in $S$ to represent these external agents, and treat these interactions in the same way as we treat interactions between the actual objects of $S$.

[4] A use-case is an interaction sequence that consists entirely of simple interactions.

The more serious problem with the answer that $I'$ is a refinement of $I$ if it implies $I$ is the following: Given our previous discussion, the elements in the sequences that $I$ is concerned with are not, in general, the same kind of elements as the ones in the sequences that $I'$ is concerned with. Rather the former correspond to what we have called high-level interactions, whereas the latter are low-level interactions (possibly *simple* interactions) that *implement* those high-level interactions. So if $I'$ were a proper refinement of $I$ and even if we ignore the problem of the last paragraph, an implication relation will in general not hold. In view of the difference between the elements in the sequences, let us use $\lambda$ to denote a general sequence that satisfies $I$, and $\lambda'$ to denote a sequence that satisfies $I'$. Note that both $\lambda$ and $\lambda'$ record all the interactions between the objects of $S$ (as well as the 'stand-in' objects that represent objects external to $S$). The difference is that the view recorded in $\lambda$ is a high-level view whereas that in $\lambda'$ is a low(er)-level view.

Let us first consider a relatively simple kind of relation between the elements in $\lambda$ and $\lambda'$. Let $\Sigma$ be the set of elements corresponding to the high-level interactions, i.e., those that can appear in $\lambda$, and $\Sigma'$ the set of elements corresponding to the low-level interactions – those that appear in $\lambda'$. A simple situation would be for the interaction represented by any given element of $\Sigma$ to be implemented by a sequence of elements of $\Sigma'$. Thus we will need a mapping of the form:

$$\rho : \Sigma \Rightarrow \Sigma'^*$$

where $\Sigma'^*$ is the set of all (finite) sequences over $\Sigma'$.

If $\rho(i) = \langle i_1, \ldots, i_n \rangle$, that means that the high level interaction $i$ is being implemented as the sequence of lower-level interactions $i_1, \ldots, i_n$. (Some of $i_1, \ldots, i_n$ may be simple interactions, others will be compound; $i$ will, of course, be a compound interaction unless $n = 1$ and $i = i_1$.) Note that there is no assumption that only the objects involved in the interaction $i$ are allowed to be participants in interactions $i_1, \ldots, i_n$. Indeed, it is not even required that all (or any!) of the participants in $i$ be involved in one or more of $i_1, \ldots, i_n$. This may seem rather strange; while it may be appropriate to allow additional objects (such as the authentication object in the client-server example of the first section) to participate in the implementation of an interaction between a given set of objects, how can an interaction that at an abstract level involves certain objects be 'implemented' at a lower level without the participation of those objects? The answer is that this is a matter of system design and the formalism should not disallow such possibilities. For instance, it may be that the nature of the system is such that this interaction is always preceded by certain other interactions involving certain other objects and hence these other objects can consult with each other to implement the required interaction without any apparent involvement of certain of the principals.[5]

Once the mapping function is given, we can see what relation must hold between the specifications $I$ and $I'$. Essentially we need to ensure that if a given sequence $\lambda'$ satisfies $I'$, then the corresponding $\lambda$ sequence will satisfy $I$, and conversely if a given $\lambda$ satisfies $I$, the corresponding $\lambda'$ must satisfy $I'$:

$$[\forall \lambda'. I' \Rightarrow \{\exists \lambda. (\lambda' = \rho(\lambda) \land I)\}] \land [\forall \lambda. I \Rightarrow \{\exists \lambda'. (\lambda' = \rho(\lambda) \land I')\}] \qquad (1)$$

where $\rho(\lambda)$ is the sequence obtained by applying $\rho$ to each element of $\lambda$ and appending together

[5]It is probably difficult to construct a simple and natural example of this type of situation, so we won't try to do so here; but our formalism, if it is to be general, must allow for such refinements even if they are unusual.

all the resulting sequences. Thus it is the natural extension of $\rho$ which is defined over elements that correspond to 'high-level interactions' to sequences of such elements.

Rule (1) is a natural generalization of the usual implication relation between a specification $I$ and its refinement $I'$ to take account of the fact that during interaction refinement high-level interactions are implemented in terms of lower level ones. (The second implication of rule (1) is needed because of our decision to require every sequence that satisfies $I$ to be an actual possible sequence of $S$; if we removed this requirement, the second implication of (1) will disappear.)

The mapping function $\rho$ and the implications in (1) are however a bit too restrictive. They require that *every time* a given high-level interaction is implemented, it must be implemented in exactly the same way as the previous time. This would prevent refinements that exploit past history in implementing particular interactions. Suppose, for instance, that a system has two objects $F_1$ and $F_2$ and that the high level requirement says that $F_1$ must send a series of files to $F_2$. If in practice it turns out, as it does in many applications, that frequently a file that $F_1$ sends to $F_2$ is a slight revision of (or even identical to) the previous file that it sent, some designers may choose to implement this by having $F_1$ send only this differential information. The high-level specification would state that $F_1$ sends a series of files, but the lower-level implementation achieves the effect much more efficiently. This is exactly the sort of thing we want to be able to deal with but the formalism we have developed so far will not allow for it, since according to the $\rho$ function, a given high-level interaction must be mapped to a fixed set of low-level interactions.

The solution is clear from the discussion above. The function $\rho$ should not be a function over the *individual* high-level interactions but rather over the high-level sequences:

$\rho : \Lambda \Rightarrow \Lambda'$

where $\Lambda$ is the set of all possible high-level traces (essentially the set $\Sigma^*$), and $\Lambda'$ the set of all possible low-level traces. Requirement (1) doesn't need any change except to note that the function $\rho$ that it refers to is this new one that maps $\lambda$ sequences directly to $\lambda'$ sequences.

This is still not general enough. As we will see from the examples in the next section, the mapping from the high-level sequences to the low-level sequences need not be one-to-one. In other words, in general we have a *relation* between elements of $\Lambda$ and $\Lambda'$ rather than a function between the two sets. Let $\rho$ denote this relation, i.e., if $\rho(\lambda, \lambda')$ holds, that means that $\lambda'$ is one of the (possibly many) low-level sequences that the high-level sequence $\lambda$ is related to. Rule (1) will correspondingly change as follows:

$$[\forall\lambda, \lambda'.\{(\rho(\lambda, \lambda') \wedge I') \Rightarrow I\}] \wedge [\forall\lambda, \lambda'.\{(\rho(\lambda, \lambda') \wedge I) \Rightarrow I\}] \tag{2}$$

It might be useful to impose some constraints on the relation $\rho$. Clearly we would have to require that for each element $\lambda$ of $\Lambda$ that there is at least one element of $\Lambda'$ that $\lambda$ is related to. It might also be reasonable to require some type of monotonicity condition – if $\lambda_1$ is a prefix of $\lambda_2$, then for each $\lambda'_1$ that is related to $\lambda_1$, there be a $\lambda'_2$ that is related to $\lambda_2$ such that $\lambda'_1$ is a prefix of $\lambda'_2$; and vice-versa. But we won't explore these conditions further in this paper (except to briefly note at the end of section 3 that even such apparently reasonable constraints may be too strong).

Before concluding this section, we should note that in a given step of *IR*, the designer is trying to achieve the intended purpose of a set of high-level sequences, in terms of a set of low-level sequences. But the 'intended purpose' is in the eye of the designer, so to speak. A different designer may well see a different purpose in the same high-level specification (we will see a simple example of this in the next section). No formalism can expect to capture the intention behind a specification. But having a precise notation (like the $\rho$ function or relation) to specify the particular mapping of high-level interactions into low-level ones called for by a particular design allows the designer to precisely record how he intends to achieve the (intended purpose of the) high-level interactions; and the verification, as required by rule (1), allows him to establish that, given the mapping called for by his design, his refinement is indeed correct. Other designers will then be able to check whether they agree with the mapping, i.e., whether it will achieve the intended purpose of the high-level interaction, as they understand it; and whether the refinement is indeed correct, given this mapping. This will lead to an improved understanding of the design, and increase our confidence in its correctness.

# 3 Some Examples

We believe that *IR* is used extensively in the design of OO systems and a study of almost any OO system will show numerous points where *IR* is used. We consider two examples in this section, and show the application of *IR* in the design of each. Our discussion of these examples is quite informal and far from complete, the main point of the section being to show the role *IR* plays in the design of these systems.

## 3.1 A Recycling System

For our first example, we consider the 'recycling system' from Jacobson [4]. The problem is to design a recycling machine that a customer can use to return recyclable objects such as cans and bottles. A customer using the machine will deposit, one at a time, items to be recycled; when he has deposited all the items, he will request a receipt that should indicate the quantity of each type of item that the customer returned (the customer can turn in this receipt to a cashier to reclaim his deposit money). In addition, there is an operator who will, at the end of each day, request a receipt totaling all the items that have been returned by various customers during the entire day.

At this level of refinement we can see that the interaction sequence will consist of three kinds of elements: `deposit-item(...)`, `print-customer-receipt(...)`, and `print-operator-receipt(...)`. But not every sequence consisting of these three types of elements is a legal sequence. We need to impose some additional conditions:

- The values printed out as a result of a call to `print-customer-receipt(...)` element, which is represented in terms of the parameters (...) of the element, should essentially be the sum of the `deposit-item(...)` elements preceding it – up to the previous `print-customer-receipt()` element.

- The element `print-operator-receipt(...)` may only appear at the end of interaction sequence. (We are assuming that an interaction sequence corresponds to a full day.)
- The values printed out as a result of a call to `print-operator-receipt(...)` element, which is represented in terms of the parameters of the element, should essentially be the sum of all the `deposit-item(...)` elements preceding it.

A more formal discussion of the example would give the precise form of the parameters of the various types of elements, and formally express the conditions above as clauses in the assertion that the interaction sequence must satisfy.

What if the recycling machine jams, perhaps because the customer inserts an item incorrectly, or because there is some other problem? There are two possible ways of dealing with this. The first approach would be to consider this a problem entirely internal to the recycling machine, to be dealt with when we actually design the machine, in particular during the implementation of the `deposit-item()` function. Perhaps the machine has built into it, some unjamming mechanisms that are invoked *internally* when a jam occurs. This would be an example of *structural refinement* and/or *procedural refinement* of the machine, rather than a refinement of the interactions between the machine, the customer, and the operator. The customer and the operator would be entirely unaware that a jam occurred or that the unjamming mechanism was invoked.

The second approach *would* involve refining the interactions between the machine, the customer, and very likely, the operator. This is the approach that [4] takes. An `alarm()` operation is provided by the operator which is invoked (by the machine) if an item is stuck. The operator then invokes an `unjam()` operation provided by the machine, which can then continue operation. (An alternative design would have been for the machine to inform the customer of the jam, and for the customer to invoke an operation provided by the operator.) Corresponding to this refinement, we need to identify the $\rho$ function (or relation) that maps interaction sequences at the higher level to sequences at this new level. In fact, in this case it is a relation rather than a function since a given high-level sequence may be mapped to many different low-level sequences that differ from each other in the number (and points) at which the machine jams and the `alarm()` operation etc. are invoked. Indeed, the relation is just that a given (high-level) sequence $\lambda$ is related to every (low-level) sequence $\lambda'$ that is identical to $\lambda$ if we remove all the `alarm()` and `unjam()` operations from $\lambda'$; in addition, the `alarm()` and `unjam()` operations in $\lambda'$ must appear in the right order and at the right locations: each `alarm()` must be followed by an `unjam()`, and each `alarm()` must appear after a `deposit()`.[6]

Next we need to specify the assertion $I'$ that the $\lambda'$ sequences must satisfy. Essentially $I'$ would say that the `alarm()` and `unjam()` operations are in the order just specified, and that if we project these elements out of $\lambda'$, the resulting sequence must satisfy the high-level assertion $I$. With this, it is easy to check the conditions required by rule (2) of the last section, thus validating

[6] One may ask, what if the machine jams again immediately after it is unjammed? Requiring that each `alarm()` operation be immediately preceded by a `deposit()` doesn't permit this. It is easy enough to do so though: require an `alarm()` operation to be preceded either by a `deposit()` or by an `unjam()` operation. The point is that having a precise specification of the mapping allows us to identify such problems easily and correct the refinement appropriately.

the correctness of this refinement.

A few further points should be made regarding this example. We started with a specification that said that the interaction sequences are made up of elements `deposit-item(...)`, `print-customer-receipt(...)`, and `print-operator-receipt(...)`. Where did this come from? In fact, we could conceive of a still higher level specification where there is a single operation `recycle-item(...)` and the interaction sequence consists of only instances of calls to this operation. It is our prior experience with recycling systems that suggested that it would be useful to provide the customer some (monetary) incentive for recycling, hence the need for the `print-customer-receipt()` operation, etc. Thus at this stage we have already gone through one level of *IR* in introducing this operation.

Moreover a designer with a different background (perhaps he comes from a land where people recycle out of concern for the environment, rather than to get their deposit money back!) may well design a system that does not have such an operation. Very likely he would still refine the interactions to take account of possible jams in the machine (unless by coincidence his land is also the mythical place where machines never jam!) Is such a refinement incorrect? While we might argue that the earlier design we sketched is likely to be more successful in some sense, there is no question the this hypothetical design is also consistent with the high-level specification (that the interaction sequence is a sequence of calls to `recycle-item()`). The goal of our formalism is to provide a notation using which designers can precisely record their refinements, and a method by which the designer can establish, given their particular mapping of higher-level interactions to lower-level ones, that their refinement is correct. Moreover, the formalism also allows the refinement to be made in several steps as would be necessary in most actual systems, rather than in one step. The advantage of recording all the intermediate steps is that another designer can more easily see the rationale of the final set of interaction sequences (which will essentially be the set of use-cases) by studying the intermediate steps. And if for some reason the final use-cases are not satisfactory, we could back up one step of the refinement at a time, rather than going back all the way to the beginning.

## 3.2 A Pair of Fax Machines

For our next example consider a pair of *fax machines* $M_1$ and $M_2$.[7] Suppose $M_1$ wishes to send a document to $M_2$. At the highest level, this could be expressed as a single action `transmit-document(...)`, the interaction sequence being a sequence of elements each corresponding to an instance of this action.

At the next level of refinement, we would probably refine this action into a sequence of three actions: Establish a connection between $M_1$ and $M_2$, send the document from $M_1$ to $M_2$, and

[7] This example was suggested by the presentation by Buhr and Casselman in their book [2] on *use-case-maps*. Use-case-maps are essentially pictorial representations of use-cases. One important advantage of use-case-maps is that they make it easier to see the causal link between the various calls to operations in various objects more easily than if we looked at, say, a textual description of the corresponding use-case. It may be interesting to develop a similar 'interaction-sequence-map' notation for pictorial representations of interaction sequences at different levels of refinement than just at the final level, i.e., at the level of the use-case.

finally break the connection between $M_1$ and $M_2$. The interaction sequence at this level will correspondingly be a sequence of these triples repeated. Defining the appropriate $\rho$ function (here it is a function) is straightforward, as is showing the required relation between the respective assertions.

What about the next level? One thing we definitely need to do is to introduce the telephone network $T$ into the picture (since faxing documents using smoke signals is not yet a well developed technology! and if it were, we would have to involve the atmosphere, the smoke signal network, etc. into the picture). Note that although $T$ was not mentioned in the high level interactions, nor is it an internal component of either $M_1$ or $M_2$, it is entering the picture in the low-level refinement. This is an example of a situation where a high-level interaction between two objects will be mediated by a third object when the interaction is refined and is typical of ***IR***. Depending on the details of the telephone system each operation can now be broken down into a sequence of actions. Thus establishing the connection may involve the sending of the area code from $M_1$ to $T$, followed by the number for $M_2$; $T$ will then send a message to $M_2$ which acknowledges and accepts the connection (picks up the phone), $T$ then sends a message to $M_1$ indicating that the connection has been set up. Note that the details of how $T$ maps the given area code and the number to the process $M_2$ are *internal* to $T$. Conceivably, $T$ might be internally organized as a collection of cooperating processes that interact with each other for this task. This refinement, being internal to $T$, is *not* an example of ***IR***; rather it is an instance of ***SR***.

Next we can refine the high-level action of $M_1$ sending the document to $M_2$, or of tearing down the connection between $M_1$ and $M_2$. We will omit the details but we should note that other designs are possible. For instance, a smart $T$ might decide to simply accept the document sent by $M_1$ and store it internally, forwarding it to $M_2$ at a later time when traffic is low. This is a different refinement, and will of course be represented by a different $\rho$. Interestingly though, this possible refinement suggests that even the requirement, on $\rho$, of monotonicity that we mentioned at the end of section 2 might be too constraining. The problem is that the smart $T$ might not send out all the documents (and other calls) in the same order that they came in. This would be a violation of monotonicity but it seems a reasonable refinement. Further analysis is needed to see how to weaken the requirement to allow for such refinements.

# 4 Discussion

Let us return briefly to the example of *client object CO* and *server object SO* of section 1. A different designer working on the same problem might refine the high-level `identify-client()` interaction into one simple interaction between *CO* and *SO*, in which *CO* just sends its name to *SO*. No passwords or encryption keys or anything like that are introduced. Is this an acceptable refinement? The answer will probably depend on who you ask. To someone with experience in security issues, this would be an obviously bad design. But our hypothetical designer, presumably a trusting soul, can argue that there was nothing about passwords in the high-level specification. The question is really one of intention. What was the intention behind the high-level specification? As we said earlier, no formalism can be expected to specify intentions; but having the designer precisely record the mapping corresponding to his particular refinement, and establish that he is satisfying the requirements of rules (1) and (2) of section 2, allow other designers to understand the design better. When they see that, according to the mapping function, the `identify-client()` is simply mapped to an action in which *CO* sends its name to *SO*, they can, without having to study the design any further, question its reasonableness.

This is an important difference between *interaction refinement* on the one hand, and *procedural* or *structural* refinement on the other. In neither *PR* nor *SR* does intention behind the high-level specification (usually) play a role. Any refinement that ensures that the high-level specification is met, is considered satisfactory. To an extent, this is not surprising. *Interaction refinement* corresponds to the design activities that go on in the earlier stages of the design. Generally we go through a series of *IR* steps starting with the very high-level specification, until we reach the set of use-cases. Then we go through one or more steps of *SR* to develop the (internal) structure of the system, and then steps of *PR* when implementing the various operations. This is not a strictly sequential process since work on the *SR* steps may suggest revisions to the previous *IR* steps, but generally this is the progression of refinements. Given that *IR*-steps are the earliest ones in the design, it is natural for most questions about the intentions behind the system to be raised and answered during the *IR*-steps.

*IR*-steps, as the reader has no doubt noticed, are what go on during what is usually called 'analysis'. It has been long realized that the activities that take place during the analysis and design phases are not unlike each other. We believe that the reason for the similarity is that both are refinements, with the added complexity in the case of analysis that questions of intention often arise. Thus one can recast software life-cycles such as Boehm's [1] *spiral model* in terms of different types of refinements. (The remark towards the end of the last paragraph that a step of *SR* may suggest a rethinking of a just-completed sequence of *IR*-steps, should perhaps be taken to mean that the spiral might occasionally loop back on itself, rather than proceeding strictly from one step to the next.) But, to reiterate our main point, having a notation that we can use to precisely specify the mapping corresponding to a particular *IR*-step, and a formal method (rules (1) and (2) of section 2) that can be used to establish that, given this mapping, the refinement step is correct, allows designers (or analysts) to understand each other's work, and to have more confidence in the correctness of the system being designed.

Before concluding, it is worth noting that the considerations in this paper, in particular the idea of *interaction refinement* are especially important in dealing with distributed objects [7]. In a system of distributed objects, much of the system's (important) activity consists of communications between the objects. A given high-level task may be refined into one of several different sequences of interactions between the objects of the system, and it is important to be able to validate these refinements. Further, the (actual) communications that take place in the system when it is finally implemented usually look quite different from the high-level (conceptual) interactions that the high-level specification is expressed in terms of. As a result, several *IR* steps may be needed to arrive at the final interaction sequences; hence it is critical to understand and validate these refinements to ensure that the final design is satisfactory.

# 5 References


1. B Boehm, A spiral model of software development and enhancement, Software Eng. Notes, vol. 11, 1986.

2. R Buhr, R Casselman, Use case maps for OO systems, Prentice-Hall, 1995.

3. JV Guttag, Notes on type abstractions, IEEE TSE, vol. 6, 1980.

4. I Jacobson, Object-oriented software engineering, Addison Wesley, 1992.

5. GT Leavens, WE Weihl, Specification and verification of object-oriented programs using supertype abstraction, Acta Informatica, vol. 32, 1995.

6. B Liskov, J Wing, A behavioral notion of subtyping, ACM TOPLAS, vol. 16, 1994.

7. N Soundarajan, Refining interactions in a distributed system, submitted to PODC '97.