# Concurrent Software Engineering

## Preparing for Paradigm Shift

Peter Grogono
Computer Science and Software Engineering
Concordia University
Montréal, Québec, Canada
grogono@cse.concordia.ca

Brian Shearing
The Software Factory
Surrey, UK
ShearingBH@aol.com

## ABSTRACT

Software systems bridge the gap between information processing needs and available computer hardware. As system requirements grow in complexity and hardware evolves, the gap does not necessarily widen, but it certainly changes. Although today's applications require concurrency and today's hardware provides concurrency, programming languages remain predominantly sequential. Concurrent programming is considered too difficult and too risky to be practiced by "ordinary programmers". However, software engineering is moving towards a paradigm shift, following which concurrency will play a more fundamental role in programming languages. We discuss some of the implications of the shift towards process-oriented programming. We outline some of the features of our own process-oriented language. Finally, we review the potential impact on software engineering and on software development processes.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features.

## General Terms

Design, Languages.

## Keywords

Concurrency, Programming Languages, Software Development, Paradigm Shift, Evolvability.

## 1. INTRODUCTION

Kuhn hypothesized that science advances by paradigm shifts [26]. During any particular period or epoch, scientists practice "normal science" within the prevailing paradigm. Anomalous results are ignored at first, but when they can no longer be ignored, they trigger first a crisis and then a *paradigm shift.* The application of Kuhn's thesis to science is controversial, but it is obviously applicable to software engineering. It is undoubtedly true that software engineering undergoes periodic paradigm shifts. Although different programming paradigms coexist during any epoch, one paradigm dominates software development methodologies [15]. The current paradigm is object-oriented programming. It is being strongly challenged by aspect-oriented programming.

There are several indications that it is time for a paradigm shift:

> "Concurrency has long been touted as the 'next big thing' and 'the way of the future,' but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software." [41]

As has happened previously, the paradigm shift will be driven by advances in programming languages. The next generation of programming languages will have to be at least as effective as current languages and they will also have to provide new features including efficient and safe concurrency. Current approaches, based on objects, threads, and locks, will not be enough:

> "We must and can build concurrent computation models that are far more deterministic, and we must judiciously introduce nondeterminism where needed. . . Threads take the opposite approach. They make programs absurdly nondeterministic and rely on a programming style to constrain that nondeterminism to achieve deterministic aims." [29]

There are various ways in which concurrency can be added to object-oriented programing, but they all add complexity to a paradigm that is already very complex. We need a paradigm that can provide the richness of objects and aspects while using concurrency as a foundation rather than adding it as a feature.

In this paper, we discuss some of the implications of the shift to process-oriented programming, outline some of the features of our own process-oriented language, and review the potential impact on software engineering and software development processes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*C3S2E* 2008 Montréal, Québec, Canada
Copyright 2008 ACM 978-1-60558-101-9/08/05 ...$5.00.

## 2. PARADIGM SHIFT

Several forces are pushing the practice of software development towards a paradigm shift.

The most obvious force is *evolving hardware*. Current programming languages and methodologies were designed to build programs to run on single processors with kilobytes of memory. Networking facilities were unusual. We are now applying these same programming languages and methodologies to build programs that run on multicore processors with gigabytes of memory. Distributed processing is the norm rather then the exception.

A less obvious but no less significant force is the *increasing complexity* of programming languages. In the overall context of software engineering, programming languages are often seen as unimportant. The idea is that if you can get the requirements, specifications, design, model, and so on right, the coding should be straighforward. In practice, the programming languages that we use have grown from simple vines to overgrown jungles replete with complex scope controls, multiple ways of exploiting inheritance, aspects to compensate for lack of structural flexibility, threads to support multiprocessing, and locks to tame the threads. There is a widespread feeling that "ordinary programmers" cannot write reliable concurrent applications. As Lee points out, the "objects with threads" model will not work [29].

A third force is the *increasing level of expectation* that we have for new software. An early word processor, for example, was a text editor with a few bells and whistles—it could print justified text, handle a few fonts in a few different sizes, and, if you were lucky, check spelling and print a table of contents. Today's word processor is a large, complex program that can handle figures, tables, high-resolution photographs, hyperlinks, and revision management, all the while supporting an elaborate GUI. Similar transformations could be described for many other applications. In particular, many applications that used to run on single processors are now expected to run on distributed or networked systems.

We have known for many years that large programs have a long lifetime, if only because it would cost too much to rewrite them. It is often said that 80% of the lifetime cost of a program goes into non-corrective maintenance. A program, like a garden, is constantly changing to reflect new demands and tastes. This suggests that we should think in terms of *growing* software, rather than constructing or building it [22]. Continuous growth creates a fourth force, the need for software that is *easy to correct, easy to adapt, and easy to refactor*.

All of these forces can be accommodated, at least to some extent, within existing frameworks of software development; however, it is becoming increasingly difficult to meet the desired goals within the current programming paradigm.

## 3. LOCATING THE PROBLEM

An important goal of software engineering is to reduce the semantic gaps between development phases. There is a large semantic gap, for example, between a structured design and an object-oriented implementation. Ideally, there should be simple mappings between specification, design, and implementation. Since requirements are determined by the problem domain, improvements imply changes to later phases. Advances in programming languages contribute to narrowing the overall gap between requirements and implementation.

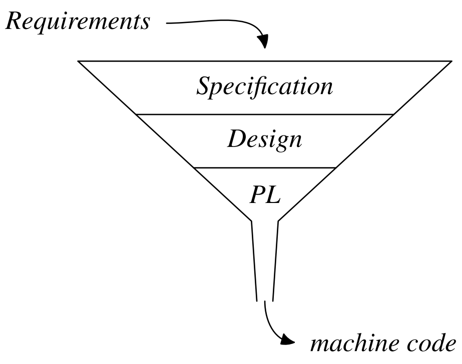

**Figure 1: The Software Development Funnel.**

Although in principle, all programming languages are capable of performing any computation, in practice, the choice of programming language has a strong effect on the ease of programming the application. Non-technical managers do not always understand this, as the following example shows. A small software company in Montreal, $S$, was recently acquired by a larger company, $L$. Most of the code developed by $S$ was written in C, and the first requirement of $L$'s management was that it be rewritten in Java because Java is "the programming language everyone uses now". While Java has many merits, it was not the most appropriate language for coding the low-level device drivers that constituted much of $S$'s code base. The management directive turned out to be an expensive mistake.

The software process can be viewed as a funnel, as shown in Figure 1. The shape of the funnel symbolizes that, while requirements range over all kinds of application, the end product of the development process, machine code, is independent of the application. Human intervention ends when the application has been coded in a programming language. The machine code is generated by compilers. As development moves down the funnel, the tools become more homogeneous.

A software development methodology covers all phases of the lifecycle. While software development itself starts with requirements elicitation and ends with implementation (moving down the funnel), new software development methodologies tend to evolve in the opposite direction, from the bottom of the funnel towards the top. The first four lines of Figure 2 show the major steps that have already occurred. Currently, aspect-oriented practices are being incorporated into development processes. If history repeats itself, the next big change—the paradigm shift—will be triggered by new programming languages. As shown in the last line of Figure 2, we hypothesize that languages based on concurrent processes will inspire process-based development. In fact, this is already happening with Rational's Rose Real-Time [14].

It is interesting to note that automatic translation from a later to an earlier paradigm is possible, but translation in the reverse direction is not possible in the general case. For example, the first implementation of C++ (AT&T's `cfront`) translated the C++ code into C code, which it then compiled. The first aspect processors were similar, in that they performed syntactic transformations to express aspects in an

| | | |
|---|---|---|
| spaghetti code | ⟶ | waterfall development |
| structured programming | ⟶ | Structured Analysis and Design Technique |
| object-oriented programming | ⟶ | Object Oriented Analysis and Design |
| aspect-oriented programming | ⟶ | Aspect Oriented Analysis and Design |
| process-oriented programming | ⟶ | Process Oriented, Model-Driven Development |

**Figure 2: Programming language evolution drives software development techniques**

object-oriented way. However, it is not possible to translate objects to aspects: the aspects cannot be reliably inferred from object-oriented code. Similarly, the process-oriented paradigm adds a distinct new level: it is not feasible to take arbitrary object-oriented code and derive an equivalent set of concurrent processes.

## 4. THE NEXT GENERATION

In order to compete in the present competitive environment, new programming languages will have to meet several requirements.

New languages must provide *safe concurrency.* This requirement has several interesting ramifications: multicore processors must be exploited efficiently; shared memory must be used for efficient communication; and facilities for distibuted programming must be available. Concurrency must be provided in such a way that "ordinary programmers" can write efficient and trustworthy concurrent programs. Safe concurrency requires compile-time checking. Consequently, it cannot be supported by external libraries and must be built into the language [6].

New languages should provide *scalability.* Most programming languages provide a fixed hierarchy of aggregating constructs (e.g., statement, method, class, and package) but applications continue to grow larger, seemingly without limit. The safest assumption that a programming language designer can make is that programs will have unbounded size. Constructs that work at all sizes must be supported.

New languages should provide *evolvability.* Programs grow, and as they grow, they evolve. We are comfortable with refactoring-in-the-small—change a pervasive name, merge a couple of methods or classes—but we cannot yet consistently manage refactoring-in-the-large—making fat clients thin by moving code to servers, or multiplexing an overloaded channel. Refactoring-in-the-large is difficult because programs contain numerous implementation details. Only by raising the level of abstraction can we make refactoring-in-the-large feasible.

*Modelling* capabilities will be required. Throughout the history of programming languages, the goal has been to reduce the semantic gap between the application domain and the source code. Modelling is the most effective way to achieve this goal.

New languages will have to facilitate *modularity* and *weak coupling* between components. Every software engineering textbook quotes the mantra "high cohesion, low coupling", invoking the spirit of Structured Design [12]. Few point out that current programming languages make loose coupling much harder to achieve than it needs to be. New approaches must ensure that program components are weakly coupled by default. Objects are coupled by control flow. Much attention is directed towards what an object provides—its public methods—but little towards what it needs. An object that provides the handful of methods needed to provide a simple service looks cohesive and would seem to promote low coupling, but if its class has twenty import declarations, it is in fact tightly coupled to many other parts of the system. If the object is moved to another processor, it will drag all its dependencies along with it.

Our central claim is that concurrent processes that communicate by passing messages will prove to be a more powerful abstraction than objects invoking one another's methods. We refer to this style as *process-oriented programming.* The idea dates back to Communicating Sequential Processes (CSP) [21], which itself was preceded by important work in the sixties and seventies [7, 13]. Jackson suggested that information systems should be modelled as concurrent processes and then transformed into procedural programs [24]. The transformation step was necessary thirty years ago but is no longer necessary today.

Although communicating processes have been studied intensively and are used in many different ways, it is nevertheless the case that no programming language based on processes exchanging messages has achieved widespread use. This might be because there are problems inherent to process-oriented programming. It could also be because process-oriented languages are not taught, or because there was a poor match between process-oriented languages and contemporary hardware, or because process-oriented languages just weren't good enough. It is also true that the object paradigm has proved to be highly effective and adaptable; success itself has reduced the need to seek other approaches.

Of the thousands of languages that have been implemented, only a handful have achieved widespread use. A somewhat larger handful have attracted small but dedicated groups of users: these are the languages that fit an ecological niche. The programming languages that have become popular have typically done so not for purely technical reasons but rather for a variety of technical and non-technical reasons: COBOL and FORTRAN were introduced by IBM, the dominant hardware vendor at the time; `C` was essential for anyone who used UNIX®; and `Java` made the internet safe. These programming languages demonstrated, in one way or another, that "worse is better".[1] Purists may be attracted by slogans such as "everything is a function" or "everything is an object", but software professionals prefer tools that get the job done.

## 5. THE ERASMUS PROJECT

Considerations such as those described in Section 4 led us to undertake a research project with the goal of designing both a programming language and a development environment. In this paper, we focus on the programming language

[1]The expression is Richard Gabriel's: see `www.dreamsongs.com/WorseIsBetter.html`.

component. The project and the language are both called Erasmus [19].[2]

Erasmus programs consist of *cells* and *processes*. Cells provide structure; processes define activities. Both are first-class entities that can be created statically or dynamically and then passed around the program. Cells may contain processes and processes may contain cells. The top-level processes in a cell (that is, those processes that are not nested inside other cells) always share a single thread of control.

A cell has multiple interfaces that explicitly determine all that the cell needs and provides. Consequently a cell can execute, or be moved into, any environment that provides matching interfaces. *Protocols*, described below, determine communication compatibility.

Processes communicate by exchanging messages and, to a limited extent, by sharing variables. In principle, this should allow Erasmus programs to exploit parallelism in processor-rich environments. To ameliorate the problems associated with concurrent programming, a process can share variables only with other processes in the same cell, and the processes in a cell execute as coroutines (i.e., non-preemptively).[3] This eliminates "nasty" race conditions, which may make shared variables inconsistent, and leaves the programmer with only a few well-defined responsibilities.

Processes communicate through *channels*. Each channel is associated with a *protocol* that determines the types of the messages that may be sent through the channel and their allowed sequences. For example, the protocol

```
[ start; *(query: Text; ^reply: Integer); finish ]
```

specifies that a client can send a signal `start` to a server, repeatedly send a `query` of type `Text` and receive (`^`) a `reply` of type `Integer` from it, and then stop the server by sending the signal `finish` to it. If the server provided two services, a clause of the form $Q; \hat{}R$ would be replaced by a clause of the form $Q_1; \hat{}R_1 | Q_2; \hat{}R_2$, in which the operator | signifies choice. Protocols do not have to match exactly; the requirement that a server protocol *satisfies* a client protocol ensures that the server can do everything that the client needs.

The compiler checks that each server satisfies its clients. Satisfaction is defined as a partial order on labelled transition systems (LTS). A LTS $(Q, L, T)$ is a labelled, directed graph with nodes (states) $Q$, edge labels, $L$, and edges (transitions) $T$. Formally, $S \sqsupseteq C$ ("$S$ satisfies $C$") if there is a mapping that preserves labels from the transitions of $C$ to the transitions of $S$. From a process or protocol, $P$, the corresponding LTS, $\mathcal{L}(P)$, can be constructed. Given a server process, $S$, its protocol $S_p$, a client process $C$, and its protocol $C_p$, the compiler constructs the respective LTS and checks that

$$\mathcal{L}(S) \sqsupseteq \mathcal{L}(S_p) \sqsupseteq \mathcal{L}(C_p) \sqsupseteq \mathcal{L}(C).$$

Constructing an LTS from a protocol is straightforward: Figure 3 shows an example. A process has an LTS corresponding to each of its ports. To construct an LTS, the process is sliced with respect to the port: that is, reads and writes to the port and associated control structures are retained, and everything else is discarded. Consequently, the compiler checks the behaviour of a process with respect to each of its ports separately. This kind of checking is useful but limited. Each linked pair of processes is checked for behaviour compatibility, but larger sets of processes are not checked—at least, not by this mechanism.

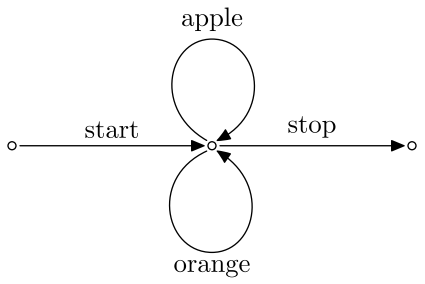

**Figure 3: LTS corresponding to the protocol `[start; *(apple | orange); stop]`.**

A process can imitate a function by providing a protocol of the form

$$[\, *(\, I_1; I_2; \ldots; I_m;\ O_1; O_2; \ldots O_n\,)\,]$$

in which the `I`s correspond to inputs and the `O`s correspond to outputs. A process can imitate an object by providing a protocol of the form

$$[\, *(\, M_1 | M_2 | \cdots | M_k\,)\,]$$

in which the `M`s specify the behaviour of the object's methods. (The choice is made by the client; the server does whatever it is asked to do.) In this way, processes and protocols can simulate both methods and objects, as well as providing many other possibilities for communication.

All data movement in an Erasmus program is controlled by assignment statements of the form `v:=e`, in which `v` is an lvalue and `e` is an rvalue. The effect is to copy either a value or a reference, depending on declarations in the current scope. Either `v`, or `e`, or both, may be port expressions of the form `p.f`. Thus `p.f:=e` sends data `e` to another process, `v:=p.f` receives data from another process and stores it in `v`, and `p1.f1:=p2.f2` is a combined send/receive, or "transfer" operation. Since receiving expressions are rvalues, programmers may write statements like `v:=p1.f1+p2.f2` to sum two inputs. The inputs may come from different regions of memory or, in principle, from different continents.

Code generation for assignment statements requires information in addition to that provided by the source code. The additional information is passed to the compiler in a separate file that defines the mapping of cells and processes to physical processors. Depending on this information, an assignment might be compiled as an in-memory transfer, a function call, or a communication over a network [27].

### *The Sleeping Barber.*

We illustrate the salient features of Erasmus with a program that implements the Sleeping Barber [13]. A village barber is normally asleep. If a customer arrives, the barber wakes up and cuts the customer's hair. The barber shop has several chairs; a customer who arrives while the barber is busy sits in a chair and waits. If there are no chairs free,

[2]This report, and other material related to Erasmus, is available at users.encs.concordia.ca/∼grogono/Erasmus-/erasmus.html.

[3]This is the semantics: an optimizing compiler is allowed to map processes in a cell to different processors provided that it inserts appropriate locks.

the customer goes away.

We use a simple protocol, consisting of just a customer"s name, for this program:

```
    prot = [ *customer: Text ]
```

The barber is represented by a process `barber` with a port `start` that receives customers who need their hair cut and a port `finish` for customers who have had their hair cut. The mode of a port is indicated by the sign of its protocol: `+prot` indicates a server and `-prot` indicates a client. The barber sleeps until a customer is received, cuts the customer's hair, and then returns the customer's name with `"had hair cut"` appended. The bar (`|`) in process `barber` separates the ports and shared variables of the process and its body.

```
  barber = { start: +prot; finish: -prot;
             sleeping: Bool |
    loop
      sleeping := true;
      customer: Text := start.customer;
      sleeping := false;
      finish.customer := customer + " had hair cut"
    end
  }
```

Most of the work is done by the process `waitingRoom`. This process uses a `loopselect` statement that repeatedly chooses an action based on a condition and a port that is ready to communicate. The keyword `random` determines the scheduling policy: if several branches of the `loopselect` statement are ready, one will be chosen at random. Other policies include `ordered` (choose the first ready branch), and `fair` (ensure that no branch is starved).

```
waitingRoom = { arrives: +prot; leaves: -prot;
         sleeping: Bool
         start: -prot; finish: +prot |
  chairs: Integer indexes Text;
  for c: Integer in 0 to 3 do
   chairs[c] := '';
  end;
  loopselect random
  |sleeping|
    start.customer := arrives.customer
  |not sleeping|
    customer: Text := arrives.customer;
    any name: Text in range chairs
                   such that name = '' do
      name := customer
    else
      leaves.customer := customer + ' went away'
    end
  ||
    leaves.customer := finish.customer;
    any name: Text in range chairs
                   such that name <> '' do
      start.customer := name;
      name := ''
    end
  end
}
```

The first branch of the `loopselect` statement is executed if `sleeping` is `true` and a customer has arrived; the customer is sent directly to the barber.

In the second branch, the barber is `not sleeping` (i.e., busy with a customer) when a customer arrives. The customer is given an empty chair, if there is one, otherwise the customer goes away.

The final branch has no condition (`||`) and is executed whenever the barber has finished with a customer. The customer leaves. If there are waiting customers, one is sent to the barber and the chair becomes free. If no one is waiting, the barber is allowed to continue sleeping.

The next component of the program is `barberShop`, which is a cell that links `barber` and `waitingRoom` and defines an interface for them. The interface of `barberShop` consists of just the two ports `arrives` and `leaves`; everything else is private, including the shared variable `sleeping`. Cell definitions are enclosed in parentheses (`(...)`) rather than braces (`{...}`) although the keywords `process` and `cell` can also be used.

```
  barberShop = ( arrives: +prot; leaves: -prot |
    sleeping: Bool := false;
    start, finish: prot;
    barber(start, finish, sleeping);
    waitingRoom(arrives, leaves,
                sleeping, start, finish)
  )
```

Next, we need two trivial processes. The first of these, `customerGenerator`, sends a stream of customers to the barber shop. The process `reporter` displays the status of customers leaving the barber shop.

```
  customerGenerator = { start: -prot |
    custnum: Integer := 0;
    loop while custnum < 20;
      custnum += 1;
      start.customer := "C" + text custnum
    end
  }

  reporter = { finish: +prot |
    loop
      sys.out := finish.customer + "\n"
    end
  }
```

The "main program" is a cell called `village` that encapsulates the processes for generating and reporting, and the cell `barberShop`.

```
  village = (
    arrives, leaves: prot;
    customerGenerator(arrives);
    reporter(leaves);
    barberShop(arrives, leaves)
  )
```

Each component that we have seen so far is a declaration. The final line of the program,

```
  village()
```

creates an instance of the cell `village`, which in turn instantiates the other cells and processes. Figure 4 shows the high-level structure of the program.

Solving the problem is an exercise in controlling access to shared variables. For example, we must guarantee that a

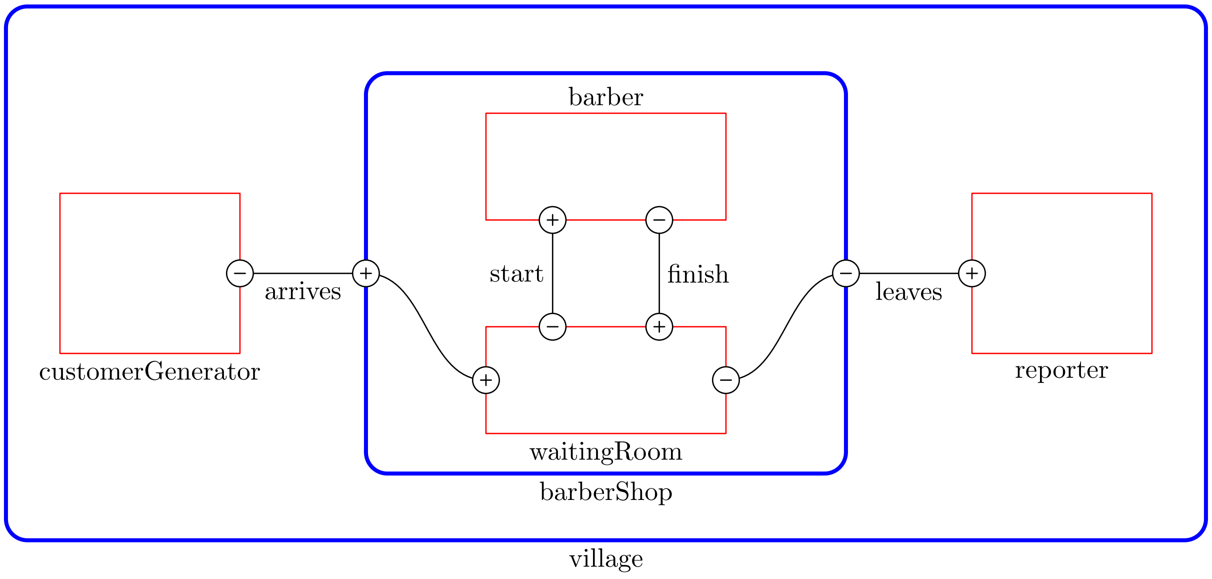

**Figure 4: The Village with a Barber Shop. Boxes with thick edges and rounded corners are cells. Boxes with thin edges and sharp corners are processes. Circles with a sign are server (+) ports or client (−) ports.**

customer is not sent to a busy barber, and that a customer is not kept waiting if the barber is sleeping. The code given respects these requirements because processes `barber` and `waitingRoom` execute as coroutines. When `waitingRoom` is active, `barber` is either sleeping and waiting for the transfer

```
    customer := start.customer
```

or not sleeping and waiting for the transfer

```
    finish.customer := customer + " had hair cut"
```

It is therefore safe for `waitingRoom` to inspect `sleeping` and act according to its value.

The coroutine convention for processes within a cell simplifies design, but it is not magic. The usual problems of concurrency are still there. In the form given above, the `barberShop` cell cannot handle customer requests of the form “Let me in only if I can be served immediately”. To service such a request, the `waitingRoom` would check that `sleeping` was `true` and then admit the customer. But it would block while waiting for the customer to come into the shop and, during this interval, the barber could take another customer. The request can be handled correctly by introducing another shared variable that is changed by `waitingRoom` and only inspected by `barber`.

## 6. MEETING THE GOALS

We now show how we expect the characteristics of Erasmus to meet the goals outlined in Section 4.

The idea of process-oriented programming is very old, but process-oriented programming has never become popular. There are a number of good reasons for this failure and, in designing Erasmus, we have tried to avoid falling into well-known traps. The first trap is the “plumbing problem”.

In object-oriented programming, connections between modules blend in with the code. Thus `o.m(d)` says “invoke method `m` of object `o` with data `d`”. In process-oriented programming, a process refers only to its ports, and connections are established by channels declared in a different part of the program. The cells `barberShop` and `village` above are simple examples, but the Sleeping Barber program has only four cells. When many processes are involved, connecting them can be tedious and error-prone: this is the plumbing problem.

Erasmus addresses the plumbing problem in several ways. Cells provide the most significant way of simplifying plumbing: they allow programs to be modularized at any scale. Appropriate cell design can simplify plumbing in the same was as the Façade pattern can simplify object-oriented designs [16].

Abbreviations could be provided to simplify coding effort, although we have not found a need to do this yet. For example, instead of writing

```
  p: prot; A(...,p,...); B(...,p,...);
```

in which “...” stands for “other ports”, we could write just

```
  A ~ B;
```

The compiler would accept this abbreviation if there was a unique protocol that could be used to link `A` and `B` and would, in effect, reconstruct the longer version.

Another problem associated with passing messages is *conversational continuity* [30]: $A$ sends a query, which forwards it to $B$, which forwards it to... $Z$. Either the reply must be sent back through the chain, $Z, Y, X, \ldots, A$, or extra channels (e.g., linking $Z$ to $A$ directly) must be installed. (In fact, the same situation occurs in object-oriented programming, too. A request sent through a chain of methods must return through each method.) Erasmus solves this problem by providing first-class ports. A process can send a port with a query; the process that eventually responds to the query uses the port to send its reply.

Early process-oriented languages had a reputation for unreliability. This was a side-effect of the plumbing problem: processes would try to send when they should be receiving; processes would send messages of the wrong type; and so on.

Erasmus checks all linkage at compile-time. A program cannot fail at run-time because a process sends when it should receive, uses the wrong type, or sends messages in the wrong order. Thus the compiler should detect most plumbing errors.

One of the most significant advantages of process-oriented programming over object-oriented programming is weak coupling. A process owns its control thread and can choose when to communicate or not communicate. An object has no control over when, or in what order, its methods are invoked. Pre/post conditions and invariants are effective reasoning tools for object-oriented programming, but much of their effectiveness evaporates in the presence of multiple threads.

Objects are linked by interfaces. Interfaces are asymmetrical, in that they describe the operations that the server can perform on behalf of the client. Although protocols refer to clients and servers, this is only to disambiguate message direction. Protocols are symmetric: either partner may provide services for the other.

Erasmus programs are *scalable*. Cells and processes may be nested to any depth. Nesting is logical, not textual. For example, a programmer might declare a process P, then declare a cell C containing P, and finally instantiate C:

```
P =  { ... };
C =  ( P; ... );
C();
```

Several features of Erasmus contribute to *evolvability*. Components are self-contained and autonomous. Their interfaces include all dependencies, a feature that also contributes to *weak coupling*. The meaning of a program (what it does) is separated from its deployment (how it works and where its processes execute). It is possible to convert a standalone program into a client-server application with little or no change to its source code [27].

Using independent components with minimal interdependencies simplifies *modelling*. The Erasmus compiler further supports modelling by accepting incomplete programs, performing as much checking as possible with the information available to it. Thus a design, consisting of cells, protocols, and process stubs, can be checked for consistency. There is a visual representation for designs, with a natural correspondence between Erasmus code and symbols in the design diagrams. Implementation consists of coding the processes; the compiler checks that process code satisfies the protocols.

By using processes as the basic abstraction, Erasmus not only provides *safe concurrency* but also makes it the fundamental programming paradigm. Because the processes within a cell execute non-preemptively, programmers do not have to worry about race conditions caused by low-level memory access. Instead, they have to deal with the more manageable problem of shared variables changing their values when a process in a cell blocks in order to communicate.

Cells and processes may be nested to any depth, making Erasmus programs *scalable*. Nesting is logical, not textual. Consequently, programs look the same at all sizes and have a "fractal" structure.

The shift to a process-oriented paradigm will affect the software development process in several ways. Environments will be expected to provide support for testing, refactoring, and metaprogramming for concurrent systems.

Although programmers were refactoring long before they had a name for it, tools for refactoring have emerged only recently. The next generation of software development environments will provide refactoring capabilities that are both broader and deeper than today's. Erasmus facilitates refactoring by *not* committing components to connect in a particular way—in other words, by abstraction.

Modern systems are often built by developing components in a general purpose programming language and then gluing them together with smaller programs written in a scripting language. Metaprogramming formalizes this technique. Erasmus recognizes the need to use more than one language to represent a complex system.

## 7. RELATED WORK

Work related to ours can be divided into two categories. Research within the current framework aims to extend object-oriented languages with abstractions for concurrency. Research directed towards a "paradigm shift" aims to bring concurrent languages into the mainstream.

Although Simula [37] introduced many of the features that we now call "object-oriented", the first language that used objects as the basic abstraction was Smalltalk [18]. Both languages provided non-preemptive concurrency. In an interview [10], Kay said:

> At some point, it just occurred to me that if you use a computer as a building block. . . that you could model every hardware component including your computer, you could model every software thing, you could get rid of data, you could get rid of procedures and you'd have a kind of a universal system building element that would model the smallest things from the largest things.

Our claim is similar, except that we see the Erasmus cell as the "universal system building element".

There were a number of early efforts to combine objects and concurrency. Enough, in fact, that the acronym "COOL" was popular for a while. Inheritance was problematic, however, leading to the identification of the "inheritance anomaly" [32]. More recently, aspects have been proposed as a way of avoiding the anomaly [34].

The Mozart Programming System [42] is based on the language Oz, which supports "declarative programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole".[4] Oz provides both message-passing and shared-state concurrency. Its designers state that message-passing concurrency is important because it is the basic framework for multi-agent systems, the natural style for distributed systems, and suitable for building highly reliable systems. It is for these very reasons that we have adopted message-passing as the basic communication mechanism in Erasmus.

Microsoft's C# relies on the multi-threaded environment .NET. There are various proposals for extending C# with concurrency primitives at a higher level of abstraction than that provided by .NET. Benton *et al.* [4] have introduced *asynchronous methods* and *chords* in Polyphonic C#. Active C# enhances C# with concurrency and a new model for object communication [20]. An *activity* is a class member that runs as a separate thread within an object. Communication is controlled by *formal dialogs.* Activities and

[4]Quoted from www.mozart-oz.org on 2008/03/15.

dialogs provide an expressive notation that can be used to solve complex concurrent problems.

UML 2 has responded to the call for concurrency by providing more flexibility than UML 1 for modeling concurrent systems. Concurrency is still modeled by forks and joins in control flows, but there is no synchronization following a fork. Following the practice of architecture description languages, system components have ports and communicate via connectors [38]. Many UML 2 models map naturally into Erasmus programs.

Typical middleware for distributed systems either attempts to hide all of the implementation details of communication (e.g., RPC) or to require programmers to do the dirty work themselves. *Infopipes* [5] provides a different approach, wherein the various aspects of communication are reified, giving programmers a collection of abstractions from which they can build customized communication systems. Infopipes are *compositional*: the properties of a channel can be inferred systematically from the components used to build the channel. For example, the result of joining two infopipes, with latencies $t_1$ and $t_2$, in series, is an infopipe with latency $t_1 + t_2$.

The other main area of research related to our project concerns process-oriented programming. The potential value of programming with concurrent processes was recognized a long time ago. The importance of loose coupling was recognized in the mid-sixties by Dijkstra and others:

> We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other. [13]

Jackson was an early proponent of concurrent processes. He considered the hardware of the time to be too primitive to actually implement concurrency directly, but he demonstrated the advantages that accrue from modelling a system as a set of processes:

> The basic form of model is a network of processes, one process for each independently active entity in the real world. These processes communicate by writing and reading serial data streams or files, each data stream connecting exactly two processes; there is no process synchronization other than by these data streams. Implementation of such a model on currently available uniprocessor or pauciprocessor machines, requires substantial transformation of the program texts. [24]

Hoare [21] introduced a theory for Communicating Sequential Processes (CSP) and Milner [35] introduced a Calculus of Communicating Systems (CCS), both systems for reasoning about communicating concurrent processes.

CSP was developed into a full-fledged programming language called Occam [33]. Occam's descendant, Occam-$\pi$ [3] has a number of features that are relevant to our project. Occam-$\pi$ demonstrates the possibility of efficient execution of many small processes; programs with millions of processes can be run on an ordinary laptop. It also provides *mobile processes*, which are processes that may be suspended, sent to another site, and resumed. Finally, Occam-$\pi$ has a formal semantics based on Milner's $\pi$-calculus [36].

Brinch Hansen designed a number of languages for concurrent programming [9]. Joyce was developed as a programming language for distributed systems [8]. The program components are *agents* which exchange messages via synchronous, typed channels. The features of Joyce that distinguished it from earlier message-passing languages such as CSP were: port variables; channel alphabets; output polling; channel sharing; and recursive agents. Static checking ensured a higher degree of security than was provided by earlier concurrent languages. Our prototype version of Erasmus is quite similar to Joyce in concept and makes use of several of Brinch Hansen's ideas. For example, `select` statements in Erasmus are similar to `poll` statements in Joyce.

In Erasmus, protocols enable extensive static checking. Protocols were inspired by *path expressions*, which were introduced for process synchronization by Campbell [11], and *Finite State Processes* (FSP), a formalism that uses finite state machines to model processes [31]. Protocol analysis verifies temporal relations between components and is therefore stronger than type checking.

*Guardians* [30] are somewhat similar to cells. Guardians consist of objects and processes and communicate by sending messages. Messages are values: addresses cannot be sent because guardians may be in different memory spaces. A guardian is intended to be an abstraction of a network node. In this sense, guardians are more concrete than cells, which are not necessarily related to network topology.

Ada [1] is one of the few industrial-strength languages that provides secure concurrency, although it is not a process-oriented language. In spite of its advantages, its use has been restricted mainly to the aerospace industry. The `select` statement of Erasmus is very similar to the "selective wait" of Ada. Ada also provides features for real-time programming, an area that we have not considered yet.

Hermes is an experimental language developed at IBM but never used in production [25, 40]. Since Hermes was designed as a system language for writing applications that might not be protected by hardware and operating system facilities, a new process is provided only with the capabilities that it needs and can obtain additional capabilities only by explicitly requesting them. Erasmus also uses a capability model not only to provide protection but also to reduce coupling.

The language now known as Erlang started life in an Ericsson laboratory as a Smalltalk program but quickly evolved into a Prolog program [2]. Erlang is now often cited as a "functional" language because individual processes are coded functionally. The goals of Erasmus are in many ways similar to those of Erlang but, out of sympathy to average programmers, we place greater emphasis on state. Erlang uses asynchronous message passing which, according to van Roy and Haridi [42, page 387], eliminates shared references between processes and maximises process independence. In Erasmus, it is easy to provide buffer processes for asynchronous communication.

The Java Application Isolation API allows components of a Java application to run as logically independent virtual machines [39]. The motivation for "isolates" is to prevent interference between components and to "simplify construction of obviously secure systems" [28].

The Singularity operating system currently under development at Microsoft Research "consists of three key architectural features: software-isolated processes, contract-based channels, and manifest-based programs" [23]. Singularity and

Erasmus share several common features, notably the emphasis on processes and on checked communications—contract-based channels in Singularity and protocols in Erasmus. Since Singularity defines contracts in terms of general state machines and Erasmus protocols are regular expressions, the two systems are formally equivalent.

## 8. CONCLUSION

The concurrency revolution is coming and the best way to address it with is a new paradigm. A new paradigm implies a new way of thinking about designing and developing software. For the transition to process-oriented programming to succeed, there will have to be greater emphasis on formal methods and compile time checking. Concurrent programming may never be easy, but it should be manageable.

Erasmus addresses concurrency by providing processes as the primary abstraction. Programs are structured in new and different ways. The process model provides a strong foundation but must be complemented by a structuring mechanism. Cells, as defined in Erasmus, are loosely coupled and amenable to reuse.

Adapting to concurrency will take time, just as it took time to adapt to object-oriented programming from structured programming. The sooner we start, the sooner we will have the tools that we need.

## Acknowledgments

The research described here was supported by the National Science and Engineering Research Council of Canada. Geoff Sharman and John Joseph Florentin asked hard questions and demanded clear answers. Sharon H. Nelson's editing skills added clarity and grace.

## 9. REFERENCES

[1] Ada. Ada 95 reference manual. Revised International Standard ISO/IEC 8652:1995, 1995. www.adahome.com/rm95. Accessed 2008/03/12.

[2] Joe Armstrong. A history of Erlang. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages*, pages 6.1–6.26, New York, NY, USA, 2007. ACM Press.

[3] Fred R.M. Barnes and Peter H. Welch. Occam-$\pi$: blending the best of CSP and the $\pi$-calculus. www.cs.kent.ac.uk/projects/ofa/kroc. Accessed 2008/03/13.

[4] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, September 2004.

[5] Andrew P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, and Calton Pu. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems*, 8:406–419, 2002.

[6] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268. ACM Press, 2005.

[7] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[8] Per Brinch Hansen. Joyce—a programming language for distributed systems. *Software—Practice and Experience*, 17(1):29–50, January 1987.

[9] Per Brinch Hansen. *The Search for Simplicity—Essays in Parallel Programming.* IEEE Computer Society Press, 1996.

[10] Mark Brunelli. Question & Answer: Smalltalk with object-oriented programming pioneer Kay. searchsoa.techtarget.com/news/interview/0,289202,-sid26_gci962762,00.html. Accessed 2008/03/25.

[11] R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, volume 16, pages 89–102. Springer, 1974.

[12] Larry Constantine and Ed Yourdon. *Structured Design.* Prentice Hall, 1979.

[13] E.W. Dijkstra. Cooperating sequential processes. Technical Report Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965. Reprinted in [17].

[14] Reedy Feggins. Designing component-based architectures with Rational Rose RealTime. http://www.ibm.com/developerworks/rational/library/-797.html. Accessed 2008/03/25.

[15] Robert W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, 1979. Turing Award acceptance speech.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[17] F. Genuys, editor. *Programming Languages (NATO Advanced Study Institute).* Academic Press, 1968.

[18] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1983, Reprinted with corrections 1985.

[19] Peter Grogono and Brian Shearing. MEC Reference Manual. Technical Report TR E-06, Department of Computer Science and Software Engineering, Concordia University, February 2008.

[20] Raphael Güntensperger and Jürg Gutknecht. Active C♯. In *2nd International Workshop .NET Technologies'2004*, pages 47–59, May 2004.

[21] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[22] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley, 1999.

[23] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Operating System Review*, 41(2):37–49, 2007.

[24] M.A. Jackson. Information systems: Modelling, sequencing and transformation. In R.M. McKeag and A.M. MacNaughten, editors, *On the Construction of Programs.* Cambridge University Press, 1980.

[25] Willard Khorfage and Arthur P. Goldberg. Hermes language experiences. *Software—Practice and Experience*, 25(4):389–402, April 1995.

[26] Thomas Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1962. Third Edition published in 1996.

[27] Nurudeen Lameed. Implementing concurrency in a processs-based language. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, March 2008.

[28] Doug Lea, Pete Soper, and Miles Sabin. The Java Isolation API: Introduction, applications and inspiration. bitser.net/isolate-interest/slides.pdf. Accessed 2007/06/14.

[29] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[30] Barbara Liskov. Primitives for distributed computing. In *SOSP '79: Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 33–42, New York, NY, USA, 1979. ACM.

[31] Jeff Magee and Jeff Kramer. *Concurency; State Models and Java Programs*. Wiley, second edition, 2006.

[32] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming language. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107Ű–150. MIT Press, 1993.

[33] D. May. Occam. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[34] Giuseppe Milicia and Vladimiro Sassone. The inheritance anomaly: ten years after. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1267–1274, New York, NY, USA, 2004. ACM.

[35] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[36] Robin Milner. *Communicating and Mobile Systems: The $\pi$ Calculus*. Cambridge University Press, 1999.

[37] K. Nygaard and O-J. Dahl. The development of the SIMULA language. In R. Wexelblat, editor, *History of Programming Languages*, pages 439–493. Academic Press, 1981.

[38] Bran Selic. What's new in UML 2.0? IBM White Paper, April 2005. Available at ftp://ftp.software.ibm.com/software/rational/web-/whitepapers/intro2uml2.pdf. Accessed 2008/03/25.

[39] Pete Soper. JSR 121: Application Isolation API Specification. Java Specification Requests. http://jcp.org/aboutJava/communityprocess/final/-jsr121/index.html. Accessed 2008/03/15.

[40] Robert Strom. *HERMES: A Language for Distributed Computing*. Prentice Hall, 1991.

[41] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, September 2005.

[42] Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2001.