# Objeto Paralelo: Um Padrão de Projeto para Programação Paralela[1]

Laís do Nascimento Salvador
Liria Matsumoto Sato

PCS- Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, travessa 3, no 158
CEP. 05508-900 - São Paulo, SP
tel: 55-11-3818-5617
e-mail: {lais,liria}@pcs.usp.br

## Resumo

O encapsulamento de dados inerente à programação orientada a objetos torna este paradigma muito atrativo na implementação de sistemas paralelos. Porém projetar um software orientado a objetos não é uma tarefa fácil e projetar software paralelo orientado a objetos é ainda mais difícil. Para facilitar o trabalho de projetar sistemas orientados a objetos, foi proposta a técnica de Padrões de Projeto, que documenta experiências bem sucedidas de projetos na área de orientação a objetos. Neste trabalho, é apresentado um padrão de projeto para programação paralela orientada a objetos, chamado Objeto Paralelo. O padrão de projeto Objeto Paralelo enfoca aspectos de paralelismo e sincronização no contexto da execução de métodos. Este padrão tem como objetivo a exploração efetiva de paralelismo com reutilização de código e programabilidade.

**Lista de Palavras-chave**: padrões de projeto, programação paralela, orientação a objetos, sincronização, programabilidade.

[1] Este trabalho foi financiado pelo CNPq e pelo projeto RECOPE/FINEP no. 3607/96

# 1. Introdução

O encapsulamento de dados inerente à programação orientada a objetos torna este paradigma muito atrativo na implementação de sistemas paralelos. Porém projetar um software orientado a objetos não é uma tarefa fácil e projetar software paralelo orientado a objetos é ainda mais difícil. Para facilitar o trabalho de projetar sistemas orientados a objetos, foi proposta a técnica de Padrões de Projeto, que documenta experiências bem sucedidas de projetos na área de orientação a objetos. Um padrão de projeto [Gam95] descreve o núcleo da solução de um problema que ocorre mais de uma vez num certo contexto, de tal forma que se possa usar esta solução sem a necessidade de reformulá-la.

Em ambientes de programação paralela um problema comumente encontrado é como obter alto desempenho e por outro lado oferecer ao usuário um bom grau de programabilidade. E visto que são sistemas difíceis de programar, é muito interessante oferecer soluções que possam ser reutilizadas em outros projetos. Outra questão importante em programação paralela são os mecanismos de sincronização e o oferecimento de formas bem definidas de expressão destes mecanismos. Quando se trata de Programação Paralela Orientada a Objetos, esta questão se torna mais crítica, resultando em problemas como a Anomalia da Herança [Mat93, Sal97], um conflito relacionado à reutilização do código de sincronização.

Neste trabalho, é apresentado o padrão de projeto Objeto Paralelo que tem como objetivo a exploração efetiva de paralelismo com um alto grau de reutilização de código e programabilidade, o qual pode ser usado na implementação de sistemas paralelos orientados a objetos.

O padrão Objeto Paralelo foi inicialmente projetado para dar suporte ao modelo de programação proposto em [Sal99]. Neste modelo o paralelismo é obtido através da execução assíncrona dos métodos dos objetos. Esta abordagem resulta em duas formas de paralelismo: paralelismo inter-objetos, paralelismo implícito à semântica do paradigma de orientação a objetos e o paralelismo intra-objeto, paralelismo entre métodos do mesmo objeto. Dessa forma, o padrão proposto pode ser usado por outros modelos de programação paralela orientada a objetos que também utilizam métodos como unidades de execução paralela. O padrão Objeto Paralelo também resolve algumas questões relacionadas à Anomalia da Herança, propondo uma técnica que separa a implementação dos objetos da implementação dos mecanismos de sincronização.

Na seção 2 é descrito o padrão Objeto Paralelo, enfocando os aspectos mais relevantes no escopo deste trabalho. O artigo apresenta apenas mais duas seções, uma de conclusões e outra de referências bibliográficas.

# 2. Objeto Paralelo: Um Padrão de Projeto

Um padrão descreve o núcleo da solução para um problema comum num determinado contexto. Ele encapsula uma forma de projeto comum e bem sucedida, usualmente uma estrutura de objetos, também conhecida como micro-arquitetura, que consiste de uma ou mais interfaces, classes e/ou objetos que obedecem um certo relacionamento e ordem [Lea00]. Dessa forma um padrão de projeto pode ser usado em outros projetos que apresentam o mesmo perfil e problemática.

A descrição completa de um padrão de projeto inclui diversos itens, entre os quais: nome, intento, motivação, solução, estrutura, participantes, colaboração, conseqüências, implementação, exemplos de código, usos conhecidos e padrões relacionados. Neste artigo serão apresentados apenas os itens mais relevantes. Uma explanação detalhada sobre padrões de projetos encontra-se em [Gam95]. Como no trabalho apresentado em [Rit97], neste artigo também é feita uma extensão ao formato de padrões, incluindo os itens **objetivos** e **avaliação**.

## *2.1 Motivação, Objetivos e Intento*

A principal motivação para o desenvolvimento do padrão Objeto Paralelo foi a necessidade do aumento de programabilidade na área de processamento paralelo. Os objetivos deste padrão são: promover desempenho, reutilização de código e programabilidade. Para atingir tais objetivos, o padrão Objeto Paralelo acopla os aspectos de paralelismo com o conceito de método, oferecendo paralelismo inter-objetos e intra-objeto. Por outro lado, este padrão propõe a separação da

implementação dos objetos da implementação dos mecanismos de sincronização, com a finalidade de aumentar a reutilização de código e a programabilidade.

### *2.2 Solução*

No padrão Objeto Paralelo, o paralelismo é obtido através de chamadas assíncronas aos métodos dos objetos. Esta abordagem resulta no paralelismo inter-objetos, que permite métodos de diferentes objetos serem executados em paralelo e no paralelismo intra-objeto, isto é, o paralelismo entre métodos do mesmo objeto.

Como os métodos podem ser executados de forma paralela, há a necessidade de se impor regras no ordenamento destas execuções. Estas regras podem indicar quando um determinado método pode ser executado ou especificar os métodos que não podem ser executados simultaneamente. Estas regras são implementadas através de mecanismos de sincronização, que neste trabalho são chamados de aspectos de sincronização [Sal99].

Os aspectos de sincronização são: Variáveis de Sincronização, Ações de Sincronização e Guardas. Estes aspectos estão associados ao conceito de método que é a unidade de execução paralela. Uma Ação de Sincronização [Neu91] é um trecho de código que deve ser executado antes (pré-ação) ou após (pós-ação) a execução de um método. O código envolvido numa ação de sincronização manipula apenas variáveis de sincronização. As Variáveis de Sincronização, como o próprio nome diz, são variáveis que guardam informações relativas aos mecanismos de sincronização. A Guarda é uma condição que habilita a execução de um determinado método, ela é usada para expressar a chamada sincronização condicional. A condição associada a uma guarda é também baseada somente nas variáveis de sincronização. Como nesta proposta há a possibilidade de paralelismo intra-objeto, é também necessária a expressão de sincronização de Exclusão Mútua para não permitir a execução paralela de métodos mutuamente exclusivos, como por exemplo, métodos que alteram os mesmos atributos de um objeto. Este mecanismo de sincronização também é previsto neste padrão através do uso de ações de sincronização e variáveis de sincronização.

### *2.3 Participantes*

Neste item são descritas as classes participantes do padrão e as suas responsabilidades. No caso do padrão Objeto Paralelo, há dois tipos de classes: as classes que devem ser adaptadas ao contexto da aplicação, denominadas **Componentes da Aplicação** e as classes que podem ser implementadas de forma independente da aplicação, denominadas **Componentes do Sistema**. Dessa forma, tem-se a seguinte configuração:

**Componentes da Aplicação:**

- **Proxy** - interface do objeto, apresenta os serviços (métodos) oferecidos pelo objeto da aplicação;
- **Servant** - provê a implementação do objeto da aplicação, isto é, a implementação dos métodos oferecidos pelo Proxy;
- **Synchronizer** - componente que provê a implementação dos aspectos de sincronização relacionados aos métodos da aplicação;
- **MethodRequest** - componente que encapsula as informações e o contexto de uma chamada a método. Este componente pode ser considerado como uma classe abstrata ou interface, que deve ser particularizada para cada método da aplicação.

**Componentes do Sistema:**

- **Scheduler** - componente responsável pela criação dos processadores virtuais e pelo enfileiramento das requisições de métodos na fila de requisições;

- **VirtualProcessor** – as instâncias da classe VirtualProcessor são responsáveis pela execução de métodos que se encontram na fila de requisição;
- **ActivationQueue** – fila de requisições de métodos, os elementos desta fila são objetos de classes que implementam a interface MethodRequest.

Convém observar que os Componentes do Sistema podem ser adaptados a uma determinada aplicação porém esta adaptação não é compulsória.

### *2.4 Colaboração*

Neste item é descrito como os componentes do padrão colaboram para executar as suas responsabilidades. O padrão Objeto Paralelo apresenta dois componentes principais: Procurador ("Proxy") que representa a interface do objeto e um Servidor ("Servant") que provê a implementação do objeto. Neste padrão, as chamadas aos métodos da aplicação são assíncronas, cada chamada a método é desvencilhada da execução do próprio método. O objeto Cliente, que é o usuário do padrão, chama métodos do objeto Procurador, estas chamadas são executadas num "thread". Por sua vez, os métodos são executados pelo Servidor em outros "threads".

Em tempo de execução, o Procurador transforma a chamada ao método do Cliente numa Requisição de Método ("MethodRequest") que é armazenada numa fila de métodos ("ActivationQueue") pelo Escalonador ("Scheduler"). O Escalonador por sua vez cria vários "threads", os chamados Processadores Virtuais ("VirtualProcessor"). Os Processadores Virtuais consultam a fila de requisições de métodos habilitados a serem executados ("executáveis") e estes métodos são executados sobre o Servidor. Numa máquina com arquitetura multiprocessadora, os Processadores Virtuais podem ser executados em processadores distintos produzindo paralelismo de granulosidade média a grossa.

Quando há necessidade de sincronização por parte dos métodos da aplicação, é criado um objeto especial da classe Sincronizador ("Synchronizer") que encapsula os aspectos de sincronização. A associação entre a classe da aplicação, Servidor, com os aspectos de sincronização é feita pelo componente MethodRequest . Neste componente há referências à implementação do próprio método como também aos aspectos de sincronização associados ao método. O componente MethodRequest apresenta a seguinte interface:

```
interface MethodRequest{

       boolean guard();
       boolean mutex_free();
       void pre_actions();
       void post_actions();
       void call();
}
```

Uma Requisição de Método faz referência a um objeto da classe Servidor, por meio do método `call()` que indica a implementação do método da aplicação e faz referências a um objeto da classe Sincronizador, por meio dos métodos:

- `guard()` – guardas que habilitam a execução do métodos;
- `mutex_free ()` – teste de exclusão mútua;
- `pre_action ()` – pré-ação de sincronização;
- `post_action ()` – pós-ação de sincronização.

## *2.5 Implementação*

Na implementação deste padrão foi usado o ambiente de programação Java [Gos96]. A linguagem Java foi escolhida por ser uma linguagem orientada a objetos e por apresentar aspectos de programação concorrente com “threads”.

Neste item são descritas, em linhas gerais, os principais componentes do sistema: o Escalonador e o Processador Virtual.

### 2.5.1 Escalonador

Uma instância da classe `Scheduler` é criada pelo componente `Proxy` da aplicação. A classe `Scheduler` é responsável pela criação da fila de requisição de métodos e pela inserção de elementos nesta fila. Outra função deste componente é a criação e ativação dos processadores virtuais.

```
class Scheduler {
       protected ActivationQueue act_queue_;
       protected Processor p[];
       private int nproc;

       private void create_procs()
       {
              ...
              for (int i=0; i<nproc; i++) p[i].start();
       }
       ...
}
```

### 2.5.2 Processador Virtual

Um objeto da classe `VirtualProcessor` executa os seguintes passos, enquanto houver requisições de métodos pendentes:

1. consulta a fila de execução de métodos;
2. verifica se o método que se encontra no início da fila pode ser executado. Esta verificação é feita por meio de consultas às regras de sincronização condicional e de exclusão mútua;
3. caso o método possa ser executado, o Processador Virtual chama os métodos associados às ações de sincronização e à implementação do método da aplicação.

```
class VirtualProcessor extends Thread{
       ...
       public void run()
       {
         while ( act_queue_.empty() == false) {
                    MethodRequest mr;
                    mr = act_queue_.begin();
                    if ( mr.guard() && mr.mutex_free())
                    {
                              mr.pre_actions();
                              mr.call();
                              mr.post_actions();
                    }
                    ...
              }
         }
         ...
}
```

## *2.6 Exemplos Resolvidos*

Neste tópico serão apresentados dois exemplos que utilizam o padrão de projeto Objeto Paralelo. O primeiro é o exemplo de um buffer circular com as operações de inserir e retirar elementos. Este exemplo mostra como são tratados os aspectos de sincronização. O outro exemplo implementa um servidor de multiplicação de matrizes. Esta aplicação foi implementada para mostrar o potencial de desempenho da proposta.

### 2.6.1 Buffer Circular

Este exemplo trata do caso clássico de um "buffer" circular com os métodos `put()` e `get()`.

A interface `BBufProxy` e a classe `BBufProxyImpl` assumem a responsabilidade da componente `Proxy`. Na classe `BBufProxyImpl` é realizada a associação com o componente do sistema `Scheduler`. Nesta classe também é criada uma instância do componente `Servant`.

Por sua vez, a classe `BBufServant` assume o papel da componente `Servant`, nota-se que esta classe apresenta a implementação usual de um buffer circular, porém não apresenta nenhum código de sincronização.

```
interface BBufProxy {

	void put(int x);
	int_Future get();
}

class BBufProxyImpl implements BBufProxy {

	protected BBufServant servant_;
	 protected BBufSynchronizer synchronizer_;
	protected Scheduler scheduler_;

	public BBufProxyImpl(int size1, int size2, int nproc) {
		servant_ = new BBufServant(size1);
		synchronizer_ = new BBufSynchronizer(size1);
		scheduler_ = new Scheduler(size2,nproc);
	}

	public void put(int x) {

		MethodRequest method_request =
				new Put (servant_,synchronizer_,x);
		scheduler_.enqueue(method_request);

	}

	public int_Future get() {
		int_Future result=0;

		MethodRequest method_request =
				new Get (servant_,synchronizer_,result);
		scheduler_.enqueue(method_request);
		return result;
	}
}

class BBufServant {
	private final int SIZE;
	private int in, out, buf[];

	public BBufServant(int bbuf_size){ SIZE = bbuf_size;
				buf  = new int[SIZE];
				in   = out = 0; }
```

```
        // Metodos da Aplicação
        public void put(int x){buf[in%SIZE] = x; in++;}
        public int_Future get(){return buf[out++%SIZE];}
}
```

As classes `Put` e `Get` implementam a interface `MethodRequest` e fazem referência ao objeto `servant_` (instância da classe `BBufServant`) no método `call()` e fazem referência ao objeto `synchronizer_` (instância da classe `BBufSynchronizer`) nos métodos associados aos aspectos de sincronização.

```
class Get implements MethodRequest {

        private BBufServant servant_;
        private BBufSynchronizer synchronizer_;
        private int_Future result_;

        Get(BBufServant servant, BBufSynchronizer synchronizer, int_Future result){
               servant_ = servant;
               synchronizer_ = synchronizer;
               result_       = result;
        }
        public boolean guard() { return synchronizer_.guard_get();}
        public void call() { result_ = servant_.get();}
        public void pre_actions() { synchronizer_.pre_get(); }
        public void post_actions() { synchronizer_.post_get(); }
        public boolean mutex_free() { return synchronizer_.mutex_free_get();}
        public void finalize() {}
}

class Put implements MethodRequest {

        private BBufServant servant_;
        private BBufSynchronizer synchronizer_;
        private int x_;

        Put(BBufServant servant, BBufSynchronizer synchronizer, int x){
            servant_ = servant;
            synchronizer_ = synchronizer;
              x_       = x; }

        public boolean guard() { return synchronizer_.guard_put();}
        public void call() { servant_.put(x_); }
        public void pre_actions() { synchronizer_.pre_put();}
        public void post_actions() { synchronizer_.post_put();}
        public boolean mutex_free() { return synchronizer_.mutex_free_put();}
        public void finalize() {}
}
```

A classe BBufSynchronizer encapsula os aspectos de sincronização associados aos métodos `put()` e `get()`. De forma resumida, estes aspectos de sincronização são:

1. O método `put()` não pode ser executado quando o buffer estiver cheio e o método `get()` não pode ser executado quando o buffer estiver vazio;
2. Os métodos `put()` e `get()` podem ser executados em paralelo, pois acessam diferentes porções do buffer, porém o método `put()` não pode ser executado em paralelo com outros métodos `put()` e o mesmo ocorre com o método `get()`.

A questão levantada no item 1 é resolvida por meio de guardas que consultam as variáveis de sincronização: `n_put` e `n_get`. Estas variáveis são alteradas pelas ações de sincronização, no caso as pós-ações: `post_put()` e `post_get()`.

O aspecto de sincronização descrito no item 2 é a necessidade de exclusão mútua entre métodos que não podem ser executados em paralelo. Esta questão é tratada através de variáveis de sincronização que indicam a execução em andamento de um determinado método. No exemplo, as variáveis para tratamento de exclusão mútua são `mutex_put` e `mutex_get`, que são alteradas pelas ações de sincronização: `pre_put()`, `pre_get()`, `post_put()` e `post_get()`. Como as ações de sincronização têm como função alterar dados compartilhados, elas são encerradas em regiões críticas.

```
class BBufSynchronizer{

        private final int SIZE;

        public BBufSynchronizer(int bbuf_size){ SIZE = bbuf_size;}

       // Variáveis de Sincronização
       protected int n_put=0, n_get=0;
       protected boolean mutex_put=false, mutex_get=false;

       // Guardas
       protected boolean guard_put() { return n_put-n_get < SIZE; }
       protected boolean guard_get() { return n_put > n_get; }

       // Pré-Ações
       protected void pre_put() {synchronized (...) {... mutex_put = true;}}
       protected void pre_get() {synchronized (...) {... mutex_get = true;}}

       // Pós-Ações
       protected void post_put() { synchronized(...) {... n_put++; mutex_put = false;}}
       protected void post_get() { synchronized (...){... n_get++; mutex_get = false;}}

       // Exclusão Mútua: put mutex put; get mutex get;

       protected boolean mutex_free_put() { return !mutex_put; }
       protected boolean mutex_free_get() { return !mutex_get; }
}
```

### 2.6.2 Multiplicação de Matrizes

Neste exemplo é implementado um servidor para multiplicação de matrizes. Na classe `MatrixUser` que faz o papel de cliente deste padrão, as matrizes sâo instanciadas e inicializadas e é criada uma instância da classe `MatrixProxyImpl`. Esta classe que por sua vez cria um servidor `MatrixServant`, responsável pela multiplicação de matrizes. Quando na chamada ao método `Multiply` da classe `MatrixProxyImpl`, este é posto na fila pelo `Scheduler`, para futura execução pelos processadores virtuais. Como na implementação deste servidor, as chamadas ao método de multiplicação não são sincronizadas, não foi necessária a implementação de um componente `Synchronizer`. Na classe `MatrixProxyImpl`, para cada chamada ao método de multiplicação de matrizes é criada uma instância de `Multiply`, uma classe que implementa a interface `MethodRequest`. Neste exemplo também é descrita a classe cliente `MatrixUser` para mostrar o uso do Padrão Objeto Paralelo.

```
class MatrixProxyImpl implements MatrixProxy {

       protected MatrixServant servant_;
       protected Scheduler scheduler_;
       public MatrixProxyImpl(int size2,int n_proc) {
              servant_ = new MatrixServant();
              scheduler_ = new Scheduler(size2,n_proc);
       }
```

```
        public void multiply(int[][] A,int[][] B,int[][] C,int size) {
            MethodRequest method_request = new Multiply (servant_,A,B,C,size);
            scheduler_.enqueue(method_request);       }
        public void signal_end() {
               scheduler_.signal_end();
        }
}

class MatrixServant {
        public MatrixServant(){ }
        // Methods
        public void multiply(int[][] A, int[][] B, int[][] C, int size)
        {
               for (int i=0; i<size; i++)
                      for (int k=0; k<size; k++)
                             for (int j=0; j<size; j++)
                                    A[i][j] += B[i][k]*C[k][j];
        }
}

class Multiply implements MethodRequest {
        private MatrixServant servant_;
        private int[][] A_,B_,C_;
        private int size_;
               ...
        public void call() {
                      servant_.multiply(A_,B_,C_,size_);
        }
               ...
}

class MatrixUser {
    public static void main(String[] args) {
        int a[][],b[][],c[][],d[][],x[][],y[][];

        // Instanciar e iniciar as matrizes
               ...
        // Instanciação de um “Proxy” para a multiplicação de matrizes

        MatrixProxy m_ = new MatrixProxyImpl(size_queue,n_proc);

        // Chmamadas ao método multiply do objeto m_

        m_.multiply(a,x,y,size_mat);
        m_.multiply(b,x,y,size_mat);
        m_.multiply(c,x,y,size_mat);
        m_.multiply(d,x,y,size_mat);
        m_.multiply(a,x,y,size_mat);
        m_.signal_end();
        }
   }
```

### *2.7 Avaliação*

Neste item é avaliado o Padrão de Projeto com relação aos objetivos propostos. Os objetivos do padrão Objeto Paralelo são: desempenho, reutilização de código e programabilidade

Para avaliar inicialmente o objetivo de desempenho foi implementado o exemplo do servidor de multiplicação de matrizes. Este exemplo foi executado numa máquina multiprocessadora Intel-Quad SC450NX MP, com quatro processadores Pentium II XEON – 400 MHz., usando o sistema operacional Red Hat Linux 6.2. O ganho obtido no exemplo de multiplicação de matrizes foi de aproximadamente 3, usando os 4 processadores da máquina. Nesta avaliação foi comparado o programa usando o padrão Objeto Paralelo, com o programa seqüencial em Java. Desempenho

semelhante é obtido com o mesmo programa em Java usando diretamente o mecanismo de escalonamento de threads da própria JVM (“Java Virtual Machine”).

Outra questão relacionada ao desempenho é a possibilidade de exploração de paralelismo entre métodos do mesmo objeto, o que pode ser interessante para aplicações com o perfil “múltiplos leitores e um único escritor”.

No exemplo da multiplicação de matrizes, embora o programa paralelo em Java e o programa paralelo usando o padrão apresentem desempenhos semelhantes, a vantagem do uso do padrão Objeto Paralelo reside na programabilidade pois o cliente não trabalha diretamente com “threads” em Java, ele apenas tem acesso à interface da aplicação e realiza chamadas aos métodos da interface.

A proposta de um componente especial que encapsula os aspectos de sincronização, resulta na separação entre a implementação dos objetos e a implementação dos mecanismos de sincronização. Esta abordagem diminui sensivelmente o efeito do emanharamento de código (“tangling phenomena”) descrito em [Lop97], aumenta a programabilidade e a capacidade de reutilização de código, resolvendo algumas questões relacionadas à Anomalia da Herança .

### *2.8 Padrões Relacionados*

O padrão Objeto Paralelo é baseado no padrão Active Object descrito em [Lav96], implementado na linguagem C++[Str86]. O padrão Objeto Paralelo apresenta algumas extensões que permitem paralelismo interno nos objetos e uma ênfase maior em alto desempenho. Além de apresentar uma estratégia diferente para tratamento de sincronização, que incrementa a reutilização de código.

## 3. Conclusão

Uma característica importante do padrão Objeto Paralelo é o uso de paralelismo interno ao objeto, pois a maioria das abordagens adota modelos baseados em objetos monitores. Consequentemente, há um problema adicional que é o tratamento da sincronização de exclusão mútua. Na nossa proposta, a especificação de exclusão mútua está integrada com os outros aspectos de sincronização, principalmente no que diz respeito à reutilização do código de sincronização. Porém há ainda uma dificuldade em oferecer este recurso de forma simples para o usuário. Um trabalho futuro é a proposta de um padrão de projeto específico para o tratamento da Exclusão Mútua.

Outra característica importante deste padrão é a sua implementação em Java, uma linguagem que além de incorporar o modelo de programação orientada a objetos, apresenta também mecanismos de programação com “threads”, mostrando que a Java se tornar uma ferramenta interessante em programação paralela, num futuro próximo.

## 4. Referências


[Gam95] GAMMA, E.; HELM, R.; JOHSON, R.; VLISSIDES, J. **Design Patterns - Elements of Reusable Object-Oriented Software.** Addison-Wesley, 1995.

[Gos96] GOSLING, J.; JOY, B.; STEELE, G. **The Java™ Language Specification**, Addison-Wesley, 1996.

[Lav96] LAVENDER, R.G.; SCHMIDT, D. C. Active Object – An Object Behavioral Pattern for Concurrent Programming. In John Vlissides, Jim Coplien and Norm Kerth, editors, **Patterns Languages of Program Design 2**, Addison-Wesley, 1996.

[Lea00] LEA, D. **Concurrent Programming in Java: design principles and patterns.** Addison-Wesley, 2000.

[Lop97] LOPES, C. V.; KICZLES, G. D. **A Language Framework for Distributed Systems**. Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center, 1997.

[Mat93] MATSUOKA, S.; YONEZAWA A. Analysis of Inheritance Anomaly in Concurrent Object Oriented Programming. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, **Research Directions in Concurrent Object-Oriented Programming.** The MIT Press, Cambridge, MA, p.107-150, 1993.

[Neu91] NEUSIUS, C. **Synchronization Actions**. In Proceedings of ECOOP'91, volume 512 of Lecture Notes in Computer Science, p.118-132. Springer-Verlag, 1991.

[Rit97] RITO, A. S. A Quality Design Solution for Object Synchronization. In Christian Lengauer, Martin Griebl and Sergei Gorlatch, editors, **Workshop 05+06: Programming Languages and Concurrent Object Oriented Programming – Euro-Par´97**, Lecture Notes in Computer Science, p. 576-580.

[Sal97] SALVADOR, L.N.; SATO, L.M. **Anomalia da Herança em Programação Concorrente Orientada a Objetos**. Anais do IX Simpósio de Arquitetura de Computadores – Processamento de Alto Desempenho, p.431-446, 1997.

[Sal99] SALVADOR,L.N., SATO, L.M.; **Synchronization Constraints in a Concurrent Object Oriented Programming Model**. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), vol. IV, pp. 2091-2094, 1999.

[Str86] STROUSTRUP, B. **The C++ Programming Language**. Addison-Wesley, Reading, MA, 1986.