Aplicação de Padrões de Concorrência em Ambientes de Desenvolvimento Distribuído

Application of Concurrence Patterns in Distributed Development Environments

Roni F. Banaszewski
Departamento de Ciência da Computação
Universidade Estadual do Centro-Oeste – UNICENTRO
ronifabio@gmail.com

Marcos A. Quináia
Departamento de Ciência da Computação
Universidade Estadual do Centro-Oeste – UNICENTRO
quinaia@unicentro.br

Elisa H. M. Huzita
Departamento de Informática
Universidade Estadual de Maringá – UEM
elisa@din.uem.br

Resumo: De modo prático, este artigo aponta a importância da aplicação de padrões de *software* para a construção de sistemas complexos, como um ambiente de desenvolvimento distribuído de *software* denominado DiSEN. Os padrões apresentados implicam a concorrência e fornecem soluções eficazes para auxiliar a produção de um sistema com qualidade e promover ganhos de produtividade à equipe de pesquisa.

Palavras-chave: padrão; concorrência; ambiente distribuído.

Abstract: In a practical way, this article points to the importance of the application of software patterns for the development of complex systems, such as the software distributed development environment named DiSEN. The presented patterns

imply concurrence, and they supply effective solutions to aid the production of a quality system and the increment of productivity to the research team.

Key words: pattern; concurrence; distributed environment.

1. Introdução

Nos últimos anos, o *software* se tornou um componente vital para os negócios. O sucesso de uma organização cada vez mais depende da utilização do *software* como um diferencial competitivo. Ao mesmo tempo, a economia tem convertido os mercados nacionais em mercados globais, criando novas formas de competição e colaboração [HER2001].

No entanto, o mercado global de *software* vem atravessando diversas crises, tanto por inúmeras falhas em projetos como por grande demanda por *software* e que, muitas vezes, agrava-se ainda mais, devido ao fato de ter disponível poucos profissionais capacitados. A saída para muitos, é a adoção da estratégia de produzir *software* remotamente, utilizando ambientes de desenvolvimento distribuído.

Este artigo refere-se à aplicação de padrões de *software* como auxílio na construção de um Ambiente de Desenvolvimento Distribuído de *Software* (ADDS) baseado em agentes denominado DiSEN (*Distributed Software Engineering Environment*). Um ADDS busca combinar técnicas, métodos e ferramentas para apoiar o engenheiro de software na construção de produtos de *software*, abrangendo todas as atividades inerentes ao processo, tais como gerência, desenvolvimento e controle da qualidade [FAL1998]. Portanto, o DiSEN, quando concluído, atuará como automatizador de todas as atividades do ciclo de desenvolvimento de *software* e unirá desenvolvedores dispersos em diferentes áreas geográficas, com intuito de gerar produtos de *software* de qualidade.

Com essa breve descrição sobre o DiSEN, percebe-se o alto grau de complexidade que seu desenvolvimento demanda. O desenvolvimento do DiSEN ao longo do seu ciclo, trata de problemas complexos como concorrência, sincronização de acesso a um objeto distribuído entre outros. Neste ponto encontra-se um grande desafio: como desenvolver um ambiente desta proporção, em bom

nível de qualidade, com uma equipe pouco experiente em assuntos supracitados e com prazo certo para entrega do produto final?

Uma alternativa que surge, com o objetivo de promover uma qualidade maior aos produtos de *software* desenvolvidos, é a adoção de padrões de *software*. Padrões são criados por profissionais especialistas em uma determinada área através do relato de suas experiências, para que estas sejam utilizadas com eficiência em projetos de desenvolvimento de *software*. Com isso, ganham os profissionais inexperientes que aproveitam da essência das soluções disponibilizadas pelos especialistas e adaptam às suas necessidades em projetos que participam, construindo assim, sistemas computacionais com alto grau de qualidade e confiabilidade.

2. Arquitetura DiSEN

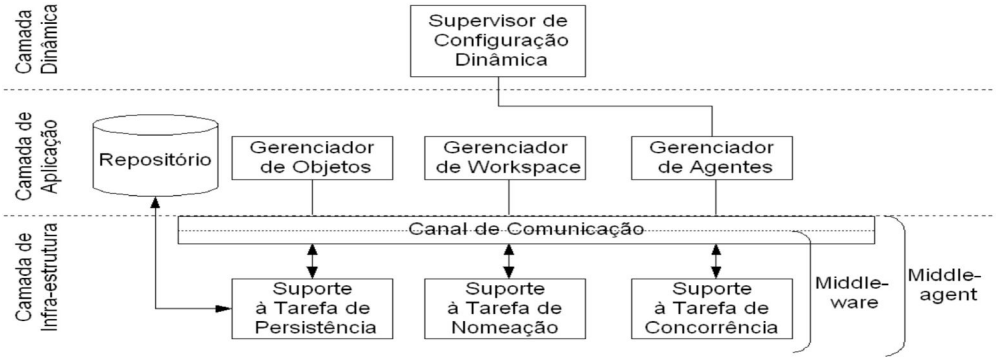
A arquitetura do ambiente DiSEN é dividida em três camadas, conforme ilustra a figura 1.

1) Camada Dinâmica: responsável pela inserção, remoção e configuração dos componentes de software e serviços em tempo de execução;

2) Camada de Aplicação: suportará um gerenciador de *workspace*, de agentes, objetos e de um banco de dados para armazenar os dados referentes ao ambiente;

3) Camada de Infra-estrutura: define o alicerce da arquitetura, provendo suporte às tarefas de nomeação, persistência e concorrência.

Figura 1. Arquitetura do DiSEN [PAS2002]



Conforme a figura 1, cada camada é subdividida em módulos específicos, como o Gerenciador de Objetos. Os padrões contidos neste artigo tratam de resolver problemas relacionados especificamente ao Gerenciador de Objetos.

O Gerenciador de Objetos é constituído por seis gerenciadores:

2.1 Gerenciador de atividades

Responsável por gerenciar atividades que fazem parte de um processo de *software*. As atividades geram ou modificam artefatos. Cada atividade está associada a participantes, ferramentas e artefatos.

2.2 Gerenciador de recursos

Gerenciará recursos necessários à realização das atividades. Os recursos podem ser materiais (computador, impressora), ferramentas (aplicativos) ou recursos humanos.

2.3 Gerenciador de Artefatos

Exemplos de artefatos são diagramas, códigos-fonte entre outros, que são criados ou modificados durante um processo. Um artefato pode servir tanto como entrada quanto saída em uma atividade.

2.4 Gerenciador de Projetos

O gerenciador é responsável por gerenciar os projetos criados, bem como as métricas e estimativas associadas a cada projeto.

2.5 Gerenciador de Processos

O gerenciador de processos analisa e verifica modelos de processo, fornece a arquitetura do produto a ser desenvolvido de acordo com as diretivas do processo.

2.6 Gerenciador de Versão e Configuração

Gerencia as várias versões pertencentes a um artefato. Estas versões devem ser gerenciadas de modo efetivo, devido a disponibilidade destes artefatos a diversos usuários que utilizam o DiSEN.

3. Aplicação de Padrões no DiSEN

A complexidade no desenvolvimento do DiSEN, em certo ponto, está relacionada à característica intrínseca do ambiente, que envolve a utilização de concorrência entre os objetos. Muitos problemas envolvendo concorrência surgem no desenvolvimento do projeto. Estes problemas referem-se à comunicação síncrona e assíncrona entre componentes, organização de funções a serem executadas concorrentemente, compartilhamento de dados entre outros.

Com o objetivo de auxiliar os desenvolvedores do ambiente DiSEN na solução de problemas encontrados e também de familiarizá-los com o uso de padrões, neste trabalho são apresentados alguns padrões encontrados em [SCH2004]. Os padrões são apropriados para solucionar problemas relativos a ambientes que envolvem concorrência por recursos computacionais. Nas seções 3.1 a 3.5 desse artigo, são apresentados os padrões e as aplicabilidades dos mesmos no DiSEN, bem como comentários sobre os resultados dos testes realizados com cada um dos padrões.

3.1 Asynchronous Completion Token (ACT)

3.1.1 Objetivo

Este padrão remete o resultado do processamento de operações para o cliente em resposta à invocação assíncrona de um serviço [SCH2004].

3.1.2 Contexto

Um sistema dirigido a eventos, onde clientes invocam operações assincronamente a serviços e subseqüentemente processam as respostas.

3.1.3 Problema

Quando uma aplicação cliente invoca uma operação assíncrona, requisitando o processamento de um ou mais serviços, cada serviço executa a requisição e remete a resposta gerada para o cliente.

A aplicação cliente deve conhecer a ação correta a ser executada em relação a cada resposta recebida concorrentemente, despendendo um tempo mínimo. Para isso, deve-se definir um mecanismo de demultiplexação para associar serviços com as suas respectivas respostas. Para tanto, é necessário solucionar os seguintes problemas:

- Quando a resposta chega para o cliente, ele deve gastar um tempo mínimo para detectar a ação e o estado cabível para a execução. Em particular, procurar em uma grande tabela é inaceitável, porque o desempenho cairia significativamente;
- É difícil para o serviço determinar qual operação o cliente deve executar, isso porque, o serviço não conhece o contexto em que o cliente invocou o método. Então, é de responsabilidade do cliente determinar quais ações e estados associar à resposta recebida;
- Deve sempre haver o menor gasto possível na comunicação entre o serviço e o cliente. Isto é de suma importância para clientes com largura de banda limitada.

3.1.4 Solução

A solução é associar as ações e estados específicos da aplicação cliente com as respostas (que indicam o resultado da operação assíncrona) enviadas pelo serviço. Isto é feito utilizando o padrão *ACT*. O *ACT* armazena unicamente as ações e estados necessários para que as respostas de serviços sejam processadas na aplicação cliente.

Quando o cliente invoca uma operação de um serviço, um *token* é criado, chamado de *ACT*. O serviço, ao receber a invocação de um de seus métodos, processa o método e passa o *token* sem nenhuma modificação para o cliente juntamente com o resultado da operação executada.

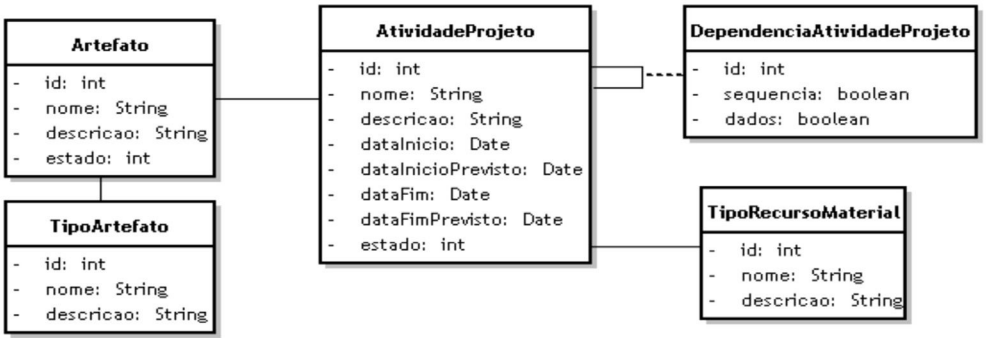
Logo, ao receber o *token*, o cliente analisa as informações armazenadas neste *token* e conclui sobre a correta ação a ser tomada em relação à resposta.

3.1.5 Aplicabilidade no DiSEN

Fazem parte do esquema do Gerenciador de Objetos do DiSEN, as seguintes classes ilustradas na figura 2:

- Atividade Projeto: indica as atividades executadas no desenvolvimento do projeto, de acordo com o processo selecionado.
- É inicialmente copiado do processo, mas o gerente de projeto tem permissão para alterá-lo;
- Tipo Recurso Material: representa os diferentes tipos de Recurso Material (papel, impressora, computador, *software*, etc);
- Artefato: representa o conjunto de arquivos usados e gerados pela execução de uma atividade.

Figura 2. Classes relacionadas com aividade projeto



No DiSEN, uma atividade necessita atender a vários requisitos para iniciar. Portanto, antes de uma atividade executar, é necessário que alguns recursos sejam alocados, assim como alguns artefatos para servir como produto para execução de suas operações.

Visto isso, percebe-se que "Atividade Projeto" precisa invocar serviços de vários objetos para executar. Assim, podem ocorrer casos em que estas invocações devem ser realizadas assincronamente, devido a inconvenientes ocorridos com os objetos responsáveis pelos serviços. Inconvenientes podem ocorrer quando algum tipo de recurso não está disponível, ou quando um artefato não está absolutamente pronto para poder ser usado como produto por outras atividades.

O uso do *Asynchronous Completion Token* resolve o problema de requisições assíncronas utilizando uma espécie de *token*. Considerando o exemplo do DiSEN, uma atividade requer algum tipo de recurso ou alguns artefatos para poder executar. Para que possa retornar um recurso para a atividade é preciso que haja, previamente, uma pesquisa para determinar se o recurso está disponível ou não naquele momento, ou seja, há um processamento. Quando o pedido do recurso é invocado, envia-se junto com a invocação um *token*, que levará consigo as informações do estado e das ações que a atividade deverá tomar quando a resposta da sua requisição retornar. Com isto não é preciso que o recurso saiba onde a sua resposta será aplicada na atividade, basta ele enviar sua resposta juntamente com o *token* de volta para a atividade, que essa saberá qual ação tomar, com base na informação contida no *token*. Isso facilita muito a comunicação entre o requerente e o serviço. No caso do DiSEN, uma atividade pode receber informações de recursos como também de artefatos necessários. A atividade deve saber como aplicar estas informações corretamente dentro de atividades através dos dados contidos no *token*. Este *token* ocupa uma pequena quantidade de espaço e por isto é viável sua utilização em ambientes distribuídos.

Esta aplicação foi testada no ambiente DiSEN, onde verificou-se a adequação perfeita deste padrão. Uma característica que impulsionou a aplicação deste padrão no DiSEN, foi a possibilidade de obter recursos de forma assíncrona. Os testes mostraram que uma atividade pode continuar executando sem precisar bloquear para esperar pelo recurso, que não é tão urgente.

3.2 *Active Object (AO)*

3.2.1 Objetivo

O padrão *Active Object* separa a invocação de um método da sua execução. Aumentando a concorrência e simplificando o acesso sincronizado a um objeto que reside no mesmo *thread* de controle [SCH2004].

3.2.2 Contexto

Clientes que acessam objetos executando em *threads* de controle separadas.

3.2.3 Problema

Quando objetos executarem concorrentemente e precisarem ser compartilhados por várias *threads* clientes, deve-se sincronizar o acesso a seus métodos e dados.

Para resolver o problema, devem-se analisar as três forças a seguir:

- Métodos invocados em um objeto que permite acesso concorrente não devem bloquear todo o processo para não prejudicar outros métodos;
- O acesso sincronizado a objetos compartilhados deve ser simples;
- Aplicações devem ser projetadas para aumentar o paralelismo disponível em plataformas de *hardware* e *software.*

3.2.4 Solução

Para cada objeto que disponibiliza execução concorrente, deve-se separar a invocação de um método desse objeto, da sua execução. Nesta tarefa, o método invocado é convertido em um objeto *Method Request* e enviado para outro *thread*, onde ele é convertido novamente em método e executado pelo objeto que implementa a interface.

3.2.5 Aplicabilidade no DiSEN

Como parte do Gerenciador de Objetos, encontram-se as classes "RecursoMaterial" e "Tarefa" ilustradas na figura 3. Estas classes possuem as seguintes descrições no DiSEN:

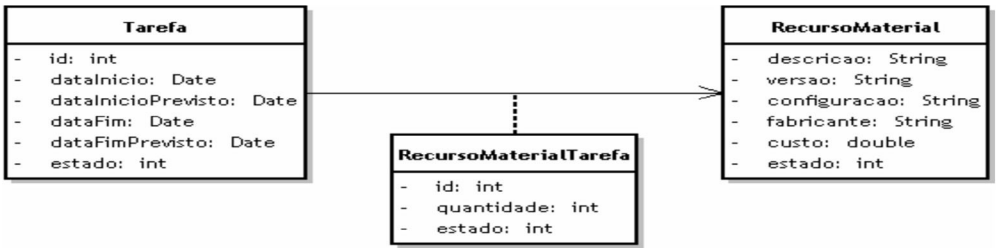
- Recurso Material: provê acesso a recursos materiais a serem utilizados pelas tarefas de um usuário;
- Tarefa*:* é uma classe associativa entre uma atividade do projeto e um participante, pois uma mesma atividade pode ser executada por vários usuários, sendo que cada tarefa necessita de recursos para executar.

Várias tarefas necessitam de recursos materiais para serem executadas e podem requerer recursos concorrentemente. O mesmo ocorre com "Recurso Material", que deve suprir as necessidades de várias "Tarefas" com seus recursos, fazendo isto eficientemente.

Considere a situação onde um "Recurso Material" precisa atender várias tarefas simultaneamente para alocar recursos. Pode ocorrer o problema de alguma tarefa estar momentaneamente impossibilitada de receber o recurso. Nesses casos, não se deve incorrer no erro de fazer esperar até que a tarefa esteja "livre" para poder receber o recurso, porque podem existir ainda *n* tarefas a serem atendidas e estas não podem ser prejudicadas pelo atraso de outras. Sendo assim, as tarefas invocarão métodos providos pela interface de "Recurso Material", referentes aos recursos a alocar. Nesta invocação, os métodos são transformados em objetos e enviados para serem executados em outro *thread*. No outro *thread* os métodos são armazenados em uma pilha e a ordem de execução é gerenciada. Na hora da invocação de um método no "RecursoMaterial", é retornado uma variável vazia para o objeto "Tarefa" que invocou. Esta variável será preenchida quando o método for processado e o resultado retornado.

Realizados os testes deste padrão no DiSEN, foi verificado a desnecessidade de bloqueio do servidor de recursos para permitir requisições. Notou-se que o servidor de recursos ou "Recurso Material" sempre esteve disponível para receber novas requisições, pois as requisições antigas estavam executando em outra *thread*, de forma assíncrona, liberando os métodos de invocação de recursos para uso por outras tarefas.

Figura 3. Classes relacionadas com recurso material



3.3 *Double Checked Locking (DCL)*

3.3.1 Objetivo

O *Double Checked Locking* tem como objetivo reduzir gastos de disputa e de sincronização toda vez que regiões críticas de um código necessitar adquirir bloqueio para executarem uma única vez e de forma concorrente [SCH2004].

3.3.2 Contexto

Instanciação de Objetos de forma concorrente.

3.3.3 Problema

Muitos padrões familiares, tais como o *Singleton* e *Iterator* [GAM1995] , trabalham bem em programas seqüenciais, mas não se adequam ao contexto de concorrência. Para esclarecimento desta situação, considere o código do *Singleton* ilustrado na figura 4.

Esta implementação do *Singleton* não trabalha bem em aplicações que requeiram preempção, multi-tarefas ou paralelismo verdadeiro. Por exemplo, ocorrem situações em aplicações que são executadas concorrentemente em que múltiplos *threads,* que executam em máquinas paralelas, invocam pela primeira vez e simultaneamente uma instância de um objeto implementado com o *Singleton*. Em casos como este, o construtor do objeto seria chamado múltiplas vezes pelos inúmeros *threads* que executam a mesma operação de criação de um objeto *Singleton*. Desse modo, a inicialização do *Singleton* violaria a propriedade de região crítica, gerando o fenômeno conhecido como condição de corrida [CHR2001] . Algumas soluções foram propostas [SCH2004], mas nem todas resolvem o problema de maneira eficaz. Na maioria delas, quando o problema de condição de corrida é resolvido ocorre o problema de *locking overhead* (gastos elevados com bloqueio) e assim sucessivamente.

Figura 4. Implementação do Singleton [SCH2004]

```
class Singleton
{
public:
static Singleton *instance (void)
{
if (instance_ == 0)
instance_ = new Singleton;
return instance_;
}
void method (void);
private: static Singleton *instance_;
};
// Chamada do método
Singleton::instance ()->method ();
```

3.3.4 Solução

"O trecho do código para resolver ambos os problemas de *locking overhead* e de condição de corrida é retratado" na figura 5.

Figura 5. Código solução [SCH2004]

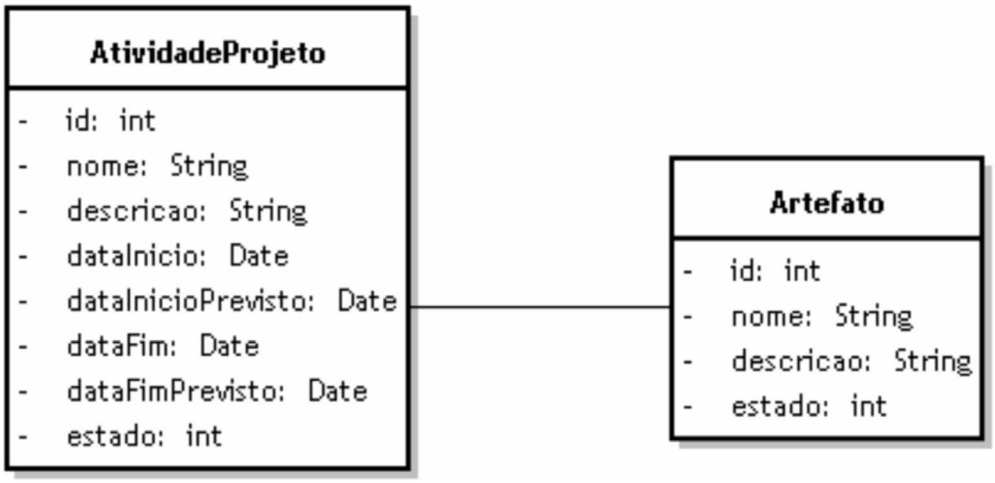
```
class DCL
{
public:
static DCL *instance (void)
{
// Primeira verificação
if (instance_ == 0)
{
// Construtor adquire o lock
Guard<Mutex> guard (lock_);
// Segunda verificação.
if (instance_ == 0)
instance_ = new DCL;
}
return instance_;
// liberação do lock
}
private:
static Mutex lock_;
static DCL *instance_;
};
```

Para solucionar o problema relatado, simplesmente duplica-se a verificação da condição, para adaptar o código do *Singleton* (Figura 4) para utilização em aplicações concorrentes. Na primeira verificação da condição, adquire-se o *lock* e só na segunda verificação é que ocorre a instanciação do objeto. A primeira verificação é útil para acabar com o problema de *locking overhead* e o segundo é para assegurar que somente um objeto foi instanciado. Assim, garante-se a inexistência de condições de corrida.

3.3.5 Aplicabilidade no DiSEN

Analisando as classes que compõem o Gerenciador de Objetos do DiSEN, encontra-se uma classe de grande importância: "Artefato" (Figura 6). A classe "Artefato" abstrai vários elementos que podem ser considerados como artefatos. Por exemplo: código ou um diagrama de casos de uso ou ainda considera-se artefato o resultado da execução de uma atividade em um projeto. Sabe-se ainda que atividades necessitam de artefatos como produtos para gerar outros artefatos.

Figura 6. Classes atividade projeto e artefato



O *Double-Checked Locking* é aplicado com sucesso para resolver o problema de instanciação que envolve um "Artefato". Para melhor esclarecer a aplicação deste padrão no Gerenciador de Objetos, considerar-se-á como exemplo prático de artefato, um diagrama de casos de uso.

No DiSEN, um artefato pode ser trabalhado concorrentemente por vários participantes. No caso de uma tarefa relacionada a uma atividade em que se tenha a construção de um caso de uso, pode-se agendar que dois ou mais participantes contribuam com esta construção. Para isso, é preciso que haja somente uma instância do artefato caso de uso sendo trabalhado pelos participantes. Isto é importante, porque os participantes alocados devem estar trabalhando na construção de um único resultado, portanto, necessitam trabalhar sobre um mesmo objeto.

O uso do padrão *Double-Checked Locking* é adequado para auxiliar a resolver problemas desta espécie. Este padrão permite a instanciação de um objeto uma única

vez, que pode ser feito concorrentemente, assim como o *Singleton* faz em aplicações seqüenciais. Além de permitir a mesma função do *Singleton* adaptado a ambientes concorrentes, ele faz isto de maneira otimizada, economizando checagem de valores.

Nos testes realizados no DiSEN, a complexidade da computação concorrente foi tratada adequadamente pelo *DCL*. Antes da aplicação do *DCL*, verificavam-se problemas ocasionados pelo inapropriado uso do padrão Singleton. Objetos implementados com o *Singleton*, circunstancialmente eram instanciados mais de uma vez, provocando caos na execução do sistema, o que alavancou uma incessante busca por uma solução adequada. Surgiu então, a solução proposta por *DCL*, eliminando o problema de instanciação duplicada.

3.4 *Thread-Specific Storage (TSS)*

3.4.1 Objetivo

Este padrão permite que múltiplos *threads* usem um ponto de acesso logicamente global para retornar um dado específico do *thread* sem causar *locking overhead* para cada acesso [SCH2004].

3.4.2 Contexto

O *TSS* deve ser empregado em aplicações *multi-threads* que freqüentemente acessam objetos que são logicamente globais e fisicamente específicos para cada *thread*.

3.4.3 Problema

Há sistemas que trabalham com variáveis que são globais, ao mesmo tempo em que seus valores devem ser específicos para cada entidade. Isto funciona bem para sistemas de único *thread*, mas quando estes sistemas são modificados para trabalharem em ambientes de múltiplos *threads*, podem ocorrer problemas como de preempção.

Preempções ocorrem quando uma variável global armazena um valor referente a uma entidade e este valor é modificado por outros *threads* que têm acesso a esta variável, tornando o valor inconsistente.

3.4.4 Solução

O *TSS* fornece algumas vantagens, tais como: aumento da eficiência, permitindo que métodos seqüenciais dentro de um *thread* acessem automaticamente um objeto de outro *thread* específico, sem que ocorra *locking overhead* para cada acesso. O *TSS* fornece também alta portabilidade e simplifica a programação.

Na figura 7, é mostrado o código exemplo em C++ da aplicação do *TSS*, onde se verifica a indiferença sobre o modo em que a aplicação executa, se em *single* ou *multi-thread*. Isso se deve a definição relatada no topo do segmento de código, para garantir a desnecessidade de mudanças no código quando o contexto da aplicação for alterado.

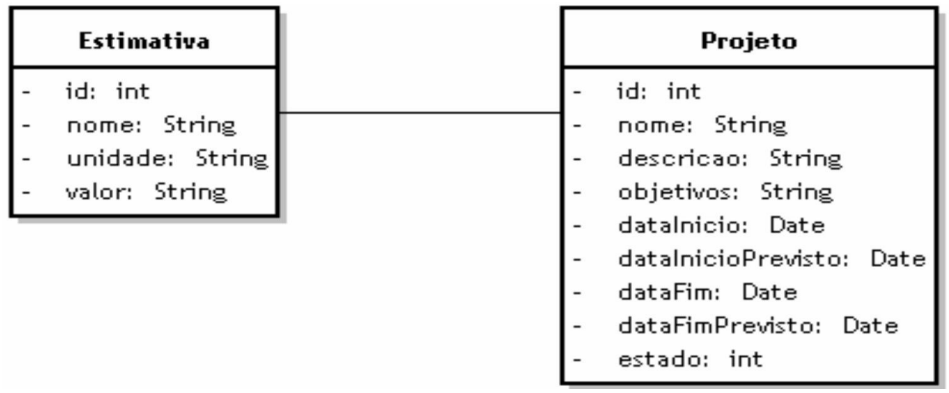
Figura 7. Código do TSS [SCH2004]

```
#if defined(_REENTRANT)
#define errno(*_errno())
#else
extern int errno;
#endif
void *worker(SOCKET socket){}
```

3.4.5 Aplicabilidade no DiSEN

No Gerenciador de Objetos, nota-se a classe "Estimativa" que se relaciona com a classe "Projeto" (relacionamento ilustrado na Figura 8). "Estimativa" é comum para todos os projetos, podendo ser considerada logicamente global e fisicamente local para cada projeto. Exemplificando: cada projeto, sem exceção, terá relação com uma classe "Estimativa" que pode ser global para todos os projetos, mas seus dados são diferentes para cada projeto. Por isso, a "Estimativa", também, necessita ser considerada fisicamente local.

Figura 8. Classes estimativa e projeto



O padrão *TSS* fornece a solução para o caso ilustrado na figura 8. Condizente aos testes realizados no ambiente DiSEN, a aplicação do *TSS* permitiu que múltiplas *threads* usassem um ponto de acesso global para retornar os dados requeridos, sem incorrer em um retardo de desempenho para cada acesso. Como proposta do padrão *TSS*, o acesso ao objeto global, "Estimativa", ocorre de maneira implícita.

3.5 *Monitor Object (MO)*

3.5.1 Objetivo

O padrão *Monitor Object* sincroniza a execução de um método para assegurar que somente um método seja executado num objeto em um tempo determinado e permite aos métodos do objeto organizar a seqüência de suas execuções [SCH2004].

3.5.2 Contexto

O *Monitor Object* é usado em aplicações onde operações de objetos são invocadas de forma concorrente por múltiplos *threads*.

3.5.3 Problema

Muitas aplicações possuem objetos que são acessados concorrentemente por vários *threads*. Para um perfeito funcionamento das aplicações concorrentes, é preciso sincronizar e organizar o acesso a estes objetos para garantir uma execução correta. Para este fim, três requerimentos devem ser satisfeitos:

- A interface de um objeto deve definir os limites da sincronização;
- Objetos, não clientes, devem ser responsáveis pela sincronização de seus próprios métodos;
- Objetos devem ser capazes de organizar seus métodos cooperativamente.

3.5.4 Solução

A solução indica alguns passos para assegurar o correto funcionamento deste padrão. Portanto, cada objeto que é acessado concorrentemente por *threads* clientes, define-se como um *Monitor Object*. Além disso, os clientes só podem acessar os serviços fornecidos pelo *Monitor Object* utilizando os métodos sincronizados. Além disso, somente um método sincronizado pode ser executado por vez, isto ocorre para evitar condições de corrida. Para obter bloqueio para um método sincronizado executar, usa-se o *Monitor Lock*. Há somente um *Monitor Lock* para cada *Monitor Object*.

A utilização deste padrão traz várias vantagens, pois simplifica a sincronização de métodos invocados concorrentemente em objetos, deixando que os próprios métodos sincronizados possam programar suas ordens de execução com auxílio dos *Monitors Conditions.* Porém, o uso do *Monitor Object* pode gerar acoplamento forte entre as funcionalidades do objeto e os mecanismos de sincronização.

3.5.5 Aplicabilidade no DISEN

Dentro do Gerenciador de Objetos do DiSEN, encontra-se um problema relacionado ao ato de adquirir e liberar algum recurso para a execução de alguma tarefa. Deve-se evitar que ocorra nestes casos de aquisição e liberação de recursos, situações em que estas operações não estejam devidamente sincronizadas, ou seja, deve haver uma política de sincronização entre os métodos adquirir e liberar, porque ambos são invocados concorrentemente por múltiplas tarefas.

Para esclarecer a idéia de aplicabilidade do *Monitor Object* como solução para o problema relatado, considerar-se-á os participantes relacionados ao gerenciamento de um recurso no DiSEN (Figura 3).

A classe responsável pela alocação de recursos será chamada de *Monitor Object* e alguns de seus métodos que necessitarem de sincronização, serão os

Synchronized Methods. No caso do Gerenciador de Objetos do DiSEN, os métodos sincronizados são adquirir e liberar, da classe ligada ao gerenciamento de recursos. O padrão *MO* trabalha com *monitors conditions*, que são condições que ocorrem dentro da aplicação para nortear o ordenamento dos métodos sincronizados, ou seja, no DiSEN isto funciona da seguinte maneira: quando uma tarefa necessitar de um recurso, que não estiver disponível no momento, esta ficará atenta a espera até que alguma outra tarefa, que adquiriu o recurso, o devolva para o repositório. Quando o recurso voltar a estar disponível, uma notificação será enviada a todas as tarefas que estão em espera.

Por meio dos testes realizados no DiSEN, observou-se a importância da aplicação deste padrão para resolver casos de notificação de eventos entre tarefas. O uso do *MO* no DiSEN, resolveu o problema do aviso instantâneo sobre a disponibilidade de um recurso a ser alocado para tarefas interessadas. Conforme os testes, verificou-se a desnecessidade de uma tarefa questionar periodicamente sobre a disponibilidade de um recurso. Com o *MO* houve uma inversão, sendo que as tarefas são notificadas imediatamente após um recurso ser liberado.

4. Conclusão

A experiência de especialistas, descritas na forma de padrões de software produzidos e testados por profissionais competentes, pode ser aplicada, mesmo por equipes com pouca experiência, a outros casos de desenvolvimento de software. Esta reaplicação do padrão possibilita aumento de produtividade e ganho de qualidade ao novo sistema construído.

O desenvolvimento de um ambiente como o DiSEN, demanda muito trabalho devido ao propósito deste ambiente executar de forma distribuída. Vários padrões foram estudados e alguns foram criados para resolver problemas recorrentes existentes no ambiente. Os novos padrões foram criados para situações onde não foram encontrados padrões prontos que resolvessem os problemas detectados. A descrição desses padrões não pertenceu ao escopo deste artigo.

Nesse artigo, foi apresentada a aplicação de apenas uma parte de todos os padrões estudados para o ambiente DiSEN. Os testes realizados no ambiente, com os

padrões relatados neste artigo, apresentaram as melhorias na resolução dos problemas encontrados. Outro benefício foi a disseminação do conhecimento entre os membros da equipe, através do aprendizado oferecido pelo uso de padrões de software.

5. Agradecimentos

Agradecemos ao CNPq pelo apoio financeiro que foi imprescindível para o desenvolvimento dos trabalhos de pesquisa. Agradecemos também às Universidades UEM e UNICENTRO pelas contrapartidas oferecidas.

6. Referências

[CHR2001] Christopher T. W. and Thiruvathukal G. K., *"Platform Computing", Multithreaded and Networked Programming*. Chapter 3: Race Conditions and Mutual Exclusion. 2001.

[FAL1998] Falbo, R. A. *Integração de Conhecimento em um Ambiente de Desenvolvimento de Software*. Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro. 1998.

[GAM1995] Gamma E., Helm R., Johnson R.,Vlissides J., *"Design Patterns:* Elements of Reusable Object-Oriented Software", Addison-Wesley Professional Computing Series, 1995.

[HER2001] Herbsleb, J. e Moitra, D. *"Global Software Development"*. IEEE Software. 5p. 2001.

[PAS2002] Pascutti, M .C. D. *"Uma Proposta de Arquitetura de um Ambiente de Desenvolvimento de Software Distribuído Baseada em Agentes"*, Porto Alegre, UFRGS (dissertação de mestrado), Agosto. 2002.

[SCH2004] Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. *"Pattern-Oriented Software Architecture"*, Patterns for Concurrent and Networked Objects. Volume 2. 2004.