LINGPON 2.0 COM SUPORTE A VETORES

Filipe Lautert, (A. F. Ronszcka), [J. A. Fabro, J. M. Simão]
PPGCA - Programa de Pós-Graduação em Computação Aplicada
Universidade Tecnológica Federal do Paraná
filipelautert@alunos.utfpr.edu.br, (ronszcka@gmail.com), [fabro@utfpr.edu.br]

***Abstract*—Este relatório descreve as definições, especificações, soluções aplicadas e implementação para a o suporte à vetores e regras associadas no Paradigma Orientado a Notificações (PON). A fim de viabilizar este suporte foram realizadas alterações na Análise Léxica e Sintática do compilador, adicionando uma nova estrutura chamada regra de formação e outras alterações para os vetores se tornarem acessíveis em qualquer parte do código. Por fim nota-se que o suporte a vetores foi uma adição bem-vinda à linguagem PON permitindo um uso mais inteligente dos seus recursos.**

***Index Terms*—Vetores, PON, C++, LingPON**

I. Introdução

Quando uma nova linguagem é introduzida, é normal que alguns de seus componentes estejam faltando ou sejam implementados de formas não ideais. Com isso a evolução e melhoria delas é algo natural a acontecer, e novas funcionalidades são adicionadas.

A LingPON foi criada para extrair o máximo do Paradigma Orientado a Notificações, e com a sua maturidade observou-se que o suporte a vetores auxiliaria na sua utilização - não só vetores, mas associando esses vetores a um novo bloco chamado regras de formação (formation_rules) que permite a geração de regras idênticas para cada índice do vetor, de modo a reduzir a quantidade de código implementado facilitando a implementação. Por exemplo, caso fosse necessário monitorar 10 sensores de movimento e enviar uma mensagem caso 1 deles abrisse, ao invés de criar 10 regras implementa-se uma única regra de formação que executará esta ação para cada sensor do vetor.

II. Contextualização

A. Linguagens e compiladores

Linguagens de programação são métodos padronizados para comunicar instruções a um computador e visam simplificar o desenvolvimento - não fosse por elas seria necessário programar diretamente em linguagem de máquina. Um artifício utilizado por estas linguagens é uma estrutura de dados conhecida como matrizes: um conjunto de variáveis do mesmo tipo acessíveis com um único nome, tendo sua individualização realizada através do uso de índices [1]. Neste relatório é abordado um tipo específico de matriz conhecido como vetor, que nada mais é que uma matriz unidimensional.

Para converter estas linguagens em código executável utilizamos compiladores. Um compilador cria um programa semanticamente equivalente, porém escrito em outra linguagem. No seu uso mais comum um compilador converte um programa de uma linguagem facilmente entendível por programadores para uma linguagem nativa de máquina (para um processador e sistema operacionais específicos). Porém esta não é a sua única forma de utilização: existem compiladores que geram código para uma máquina virtual que depois será interpretado por uma máquina virtual nativa; e compiladores que convertem o código de uma linguagem de alto nível para outra, sendo que estes compiladores também são chamados de tradutor, filtro ou conversor de linguagem [2] - em inglês são utilizados os termos source-to-source compiler, transcompiler or transpiler. É neste último tipo de compilador que estamos trabalhando e que compila a LingPON para os mais diferentes tipos de linguagens implementados.

Neste compilador é utilizada a ferramenta Lex [1] para realizar a análise Léxica do código fonte - que nada mais é que o processo de analisar o arquivo fonte e produzir uma sequência de símbolos que podem ser manipulados mais facilmente pelo analisador sintático. Após isto a ferramenta Bison [2] realiza a interpretação (análise sintática) das informações produzidas pela análise léxica, realizando chamadas à classe NOPGraph preparando o grafo da aplicação.

Este grafo será utilizado pelos geradores de código de modo a produzir o código fonte na plataforma alvo - como por exemplo código C, C++, Java, C#, etc.

B. PON e LingPON

O Paradigma Orientado a Notificações (PON) visa resolver deficiências encontradas nos paradigmas de programação mais comuns como o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD) [4] [5] [6], unificando suas melhores características visando resolver as suas deficiências. Algumas desses qualidades seriam:

- representação do conhecimento em regras
- flexibilidade de expressão
- nível apropriado de abstração

No PON, o fluxo de execução ocorre em função da mudança de estado de um atributo de um respectivo Fact Base Elements (FBE). Com esta mudança de estado são notificadas as premissas pertinentes para que reavaliem seus estados lógicos. Se este valor se altera, a premissa notifica a um ou mais conjuntos de condições conectadas sobre essa mudança de estado. Com esta notificação, a condição notificada avalia o seu

[1]https://www.gnu.org/software/flex/, visitado em 13/09/2018
[2]https://www.gnu.org/software/bison/, visitado em 13/09/2018

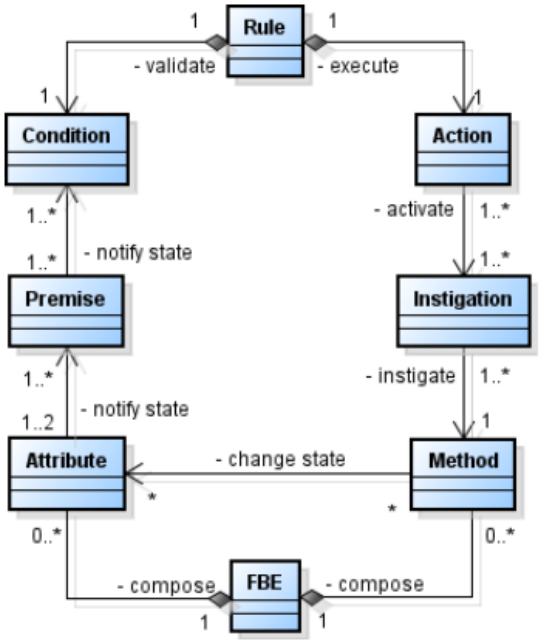

Fig. 1. Entidades do PON e seus relacionamentos [4]

valor lógico de acordo com as notificações da premissa e com o operador lógico utilizado. Caso as avaliações necessárias sejam aprovadas, a regra será informada desta aprovação [4]. Finalmente, esta regra aprovada irá ativar a sua ação.

Uma ação possui uma ou mais instigações, que irão acionar a execução de um método. Geralmente, as chamadas para os métodos mudam os estados dos atributos e o ciclo de notificação recomeça, conforme diagrama apresentado na figura 1. [4].

Para materializar tal funcionamento foi especificada a LingPON, que nada mais é que uma linguagem que possui todos estes elementos e permite a programação diretamente neste paradigma.

III. Requerimentos

Apesar da LingPON (Liguagem de programação PON) possuir todos os elementos da arquitetura descritos anteriormente implementados, observou-se que o suporte à vetores permitiria formas mais inteligentes e dinâmicas de utilizá-la associado ao novo paradigma. A utilização de vetores junto com uma instrução que permita a repetição de regras para diferentes indíces (regra de formação) facilitaria o desenvolvimento e diminuiria a repetição de regras semelhantes. Por exemplo em um caso onde houvessem muitos sensores a serem monitorados, onde caso algums seja ativado uma mensagem seja enviada com a notificação. Sem o uso de vetores é necessário criar uma regra para cada sensor, e com isso teríamos *n* regras com a mesma ação. Utilizando vetores e a regra de formação podemos escrever um único bloco de código que será repetido *n* vezes pelo compilador, uma vez para cada sensor (índice).

Baseado neste entendimento foram levantados os seguintes requisitos para a implementação de vetores:

- Suportar vetores de Atributos (Integer, String, boolean e char).
- Suportar vetores de FBE.
- Vetores com valores de índices dinâmicos (vetor[i]) e estáticos (vetor[3]).
- Vetores com valores de índices calculados (vetor[i + 1]) suportando soma e subtração.
- Nova estrutura para regras de formação: formation_rule , que define uma regra a ser repetida *n* vezes para um ou mais índices, de acordo com as estruturas *index* declaradas na mesma.
- Validação de índices dos vetores (impedir uso de índices inexistentes)

Como a linguagem já possui implementação de tradução para várias linguagens, foi decidido que não seria criada uma nova estrutura no seu grafo mas sim que os vetores seriam traduzidos para as estruturas já existentes. Sendo assim cada elemento do vetor é convertido em um atributo/instância não indexado, da seguinte forma: um vetor de nome *meuVetor* e tamanho 2, será convertido para as variáveis *meuVetor__vector_0* e *meuVetor__vector_1* , utilizando o mesmo tipo da declaração. Desta maneira o grafo gerado mantem-se o mesmo para os geradores de código existentes sem necessidade de alterações nos mesmo. Esse conversão também é realizado na hora do acesso, onde ao acessar o índice 1 do *meuVetor* a tradução é realizada para *meuVetor__vector_1*.

As regras de formação são convertidas em *n* novas regras, dependendo da quantidade de índices declarados. Dentro das formation_rules são contindos os mesmo elementos que em uma regra, assim para cada regra gerada novas referências dos filhos são gerados também (não são reaproveitados, cada regra tem suas próprias conditions, actions, etc).

IV. A linguagem

A. Declaração de vetores

A declaração dos vetores dar-se-á no seguinte formato:

- Para FBE's, utilizar <escopo> <FBE>[Quantidade] nomeFbe .
- para atributos, utilizar <escopo> <TipoAtributo>[Quantidade] nomeFbe = {valores} . A declaração dos valores para atributos é opcional.

Para os exemplos a seguir, utilizamos um FBE previamente declarado com o nome de "Alarm". Seguem exemplos de declaração:

```
# declaração de vetor de fbe Alarm com 2 Alarm
public Alarm[2] sensorVector

# declaração de vetor de fbe Alarm com 5 Alarm
public Alarm[5] alarmVector

# declaração de vetor de atributo boolean com
# 2 boolean não inicializadas
public boolean[2] xAlarm
```

```
# declaração de vetor de atributo boolean com
# 2 boolean inicializados, onde o atributo de
# índice 0 recebe true e o atributo de
# índice 1 recebe valor false
public boolean[2] jAlarm = {true, false}
```

B. Utilização em estruturas estáticas

Vetores podem ser utilizadas em qualquer parte do código, mas deve-se observar que em blocos de código não contidos em uma regra de formação o acesso precisa ser realizado diretamente a um índice do vetor, pois estando fora de uma regra de formação não temos acesso a um índice que seria utilizado para acessar as variáveis. No exemplo a seguir, ao invés de utilizar um índice dinâmico (como por exemplo, índice "i" ou "j") é utilizado um índice fixo (neste caso valores 0 ou 1):

```
# Atribuição de valor
private method mtSendSms
        attribution
            this.jAlarm[1] = true
        end_attribution
    end_method

# Utilização em uma rule
rule rlNormal
condition
    premise prThird
        this.jAlarm[1] == true
    end_premise
    or
    premise prOne
      sensorVector[0].atStatus ==
       alarmVector[0].atStatus
    end_premise
```

Utilizar um índice como *alarmVector[i].atStatus* fora de uma regra de formação irá resultar em erro de compilação.

C. Regras de formação

As regras de formação são blocos semelhantes ao bloco de regras, porém elas possuem um subitem adicional: a declaração de índices (index) . Os índices irão determinar os valores utilizados por suas variáveis durante a geração das regras. A sintaxe das regras é da seguinte forma:

- o bloco inicia com a expressão *formation_rule* e é finalizado com *end_formation_rule*
- os índices são declarados da seguinte forma: *index* <variavel> *from* <valor_inicial> *to* <valor_final>

Dentro do bloco da formation_rule deve haver ao menos uma expressão de índice, caso contrário não é possivel criar regras. Considere o código a seguir:

```
formation_rule rlFireAlarm
    index i from 0 to 1
    index s from 0 to 1
    condition
    premise prOne
        sensorVector[i].atStatus ==
         alarmVector[s+1].atStatus
    end_premise
    or
    premise prSecondFRule
        this.jAlarm[s] == true
    end_premise
    end_condition
end_formation_rule
```

Notar a utilização dos índices diretamente (sensorVector[i]) e através de operação matemática (alarmVector[s+1]) . Neste exemplo são declarados 2 índices: i de 0 à 1 e s também de 0 à 1. Neste caso teremos a geração de 4 regras (2 * 2), pois teremos a conbinação de todos os elementos de i com s. Assim serão geradas as regras:

- rlFireAlarm_i_0_s_0 substituindo *i* por **0** e *s* por **0**
- rlFireAlarm_i_0_s_1 substituindo *i* por **0** e *s* por **1**
- rlFireAlarm_i_1_s_0 substituindo *i* por **1** e *s* por **0**
- rlFireAlarm_i_1_s_1 substituindo *i* por **1** e *s* por **1**

Desta maneira o primeiro bloco de código gerado será:

```
rule rlFireAlarm_i_0_s_0
    condition
    premise prOne_i_0_s_0
        sensorVector__vector_0.atStatus ==
         alarmVector__vector_1.atStatus
    end_premise
    or
    premise prSecondFRule_i_0_s_0
        this.jAlarm__vector_0 == true
    end_premise
    end_condition
end_formation_rule
```

No código gerado fica bem claro a substituição dos índices por variáveis, de forma a permitir o entendimento pelos geradores de códigos atuais.

Caso fosse adicionado uma novo índice k com valores de 0 a 2, teríamos a criação de 12 regras (2 * 2 * 3) , e assim por diante. Desta maneira nota-se a facilidade de expandir a qauntidade de regras geradas de forma a se aproveitar melhor o potencial da linguagem.

V. Implementação

Por se tratarem de novas funcionalidades na linguagem, foi necessário realizar alterações desde a adição de novos tokens ao Flex, passando por sua interpretação e adição ao grafo no Bison, para por fim lidar com eles na classe principal do compilador, a NOPGraph . Também foi gerada uma aplicação de exemplo que contempla todas as funcionalidades implementadas, e a mesma foi adicionada ao repositório do projeto para referência.

A. Alterações no Flex

Foram declarados os novos itens para a linguagem:

- expressões *index, from* e *to*
- expressões *formation_rule* e *end_formation_rule*
- operadores + e -
- expressões *[ ]* e *{ }*

B. Alterações no Bison

No Bison foi necessário:

- Adicionar suporte aos novos tokens definidos no Flex ao Bison e na estrutura PROGRAM
- Criar os tokens *vector_instance*, *vector_attribute* e *array_factor* para suportar a declaração dos vetores de FBEs e atributos
- Adicionar os tokens *formation_rule*, *rule_indexes* e *rule_index* para interpretação do bloco das regras de formação e seus índices
- Criar o token *ID_or_VectElem* para a utilização no token *element* de modo a suportar o acesso aos vetores dentro dos blocos de códigos. Este token implicou na criação de mais alguns tokens para chegar até ele, sendo eles: *VectElem*, *vector_operation* e *operation*
- Também foi criado a classe VectorDto no C++ para possibilitar transitar a informação de vetores dentro do Bison do token VectElem até *element* .

C. Alterações na classe NOPGraph

Na classe NOPGraph - classe principal do compilador responsável pela geração do grafo - foram adicionados vários métodos para lidar com os vetores. A principal diferença com o código original é que nas regras geradas diretamente a partir de uma regra os elementos recebidos do Bison são usados diretamente, e nas regras de formação são utilizadas como modelos para a geração das regras e elementos definitivos pois é necessário instanciar novas regras e todos os outros elementos contidos nelas. Este código pode ser refatorado e externalizado em uma nova classe de modo a aumentar a modularização da compilador. São eles:

- **std::vector<Factor*>* addToFactorArray(Factor* factor)** - adiciona um valor à declaração de valores de um atributo - no linguagem representa os valores entre chaves. Ex: { 3, 4, 5}
- **std::string createVectorName(std::string name, int position)** - traduz um nome de variável para a sua versão final em vetor. Ex: *createVectorName('vec', 1)* irá resultar em *vet__vector_1* .
- **std::list<Instance*>* createInstances(Visibility *visibility, std::string fbeName, std::string name, int quantity)** - declara as instâncias de FBEs de um vetor, utilizando o método createVectorName para gerar uma entrada para cada elemento.
- **std::list<Attribute*>* createAttributes(Visibility *visibility, Type *type, std::string name, std::vector<Factor*> *values, int quantity)** - declara as instâncias de atributos de um vetor, utilizando o método createVectorName para gerar uma entrada para cada elemento. Caso sejam declarados valores entre chaves, valida se existe a quantidade correta de valores.
- **char *convertVectorToId(std::string id, int position)** - proxy para o método *createVectorName* que retorna um char* ao invés de std::string de modo a ser utilizado no Bison.
- **VectorDto *convertVectorToVectorElement(std::string id, Expression *expression)** e **VectorDto *convertVectorToVectorElement(std::string id, std::string position)** - criam uma instância de um VetorDto a partir dos dados informados. É utilizado no Bison para transmitir valores até a conversão e interpretação para Factor.
- **std::string addFormationRuleIndex(std::string id, int from, int to)** - invocado pelo Bison, adiciona um indíce a regra de formação sendo processada.
- **std::string createFormationRule(std::string name)** - invocado pelo Bison ao final da regra de formação e inicia todo o processamento para a validação e criação das rules.
- **std::string createRuleFromFormationRule(FormationRuleIndex *fri, std::string name)** - chamado à partir da createFormationRule(), processa os índices de forma recursiva e inicia a criação das rules.
- **Rule* createVectorRule(std::string name)** - executa a criação das regras e todos os seus itens agregados.
- **Condition* createVectorCondition(std::string name)** - cria as novas condições para uma rule gerada à partir da formation rule.
- **std::map<std::string, Premise*> * clonePremisseArray(std::map<std::string, Premise*> *original, std::string name)** - clona um array de premissas para ser utilizado em uma nova regra. A clonagem é realizada pois a estrutura da regra de formação é replicada para cada índice.
- **Factor *createElementFactorForClonedPromise(Factor *factor)** - invocado pela clonePremisseArray() paa criar os seus elementos.
- **std::string evaluateVectorOrVariableName(std::string name, std::string index, Expression *expression)** - utilizada para criar os novos elementos, este método valida se uma variável é um vetor e se deve ter os valores de índice interpretados ou calculados. Ex: valida se a variável é *alarmSensor*, *alarmSensor[i]* ou *alarmSensor[i+1]* e a traduz para a versão convertida caso seja um vetor - ou seja, *alarmSensor__vector_0*.
- **Action* createVectorAction(std::string name, Execution *execution, std::map<std::string, Instigation*> *instigations)** - instância uma nova action. Chamado à partir da criação de um a nova regra.
- **std::map<std::string, Instigation*>* cloneInstigationArray(std::map<std::string, Instigation*> *original, std::string name, Execution *execution)** - clona o array de instigações para ser utilizado na nova regra.
- **Instigation* createVectorInstigation(std::string name,**

Execution *execution, std::list<Call*> *calls) - cria uma nova instigação para ser utilizada no vetor de instigações clonado.

D. Novas classes

Além das alterações na classe NOPGraph, foram criadas as seguintes classes:

- **FormationRule** - utilizada para armazenar as estruturas da regra de formação durante sua interpretação pelo Bison
- **FormationRuleIndex** - armazena a informação dos índices declarados na regra de formação, sendo que um índice sempre tem um ponteiro para o próximo de maneira a possibilitar a sua utilização recursiva no método *NOPGraph.createRuleFromFormationRule()*
- **VectorDto** - classe utilizada para transmitir os dados dentro do Bison até ser possível a criação de um VectorElementFactor.
- **VectorElementFactor** - classe que estende ElementFactor e adiciona a possibilidade de transmitir dados de vetores pela estrutura do compilador.

Também foram realizadas pequenas alterações nas classes Attribute, ElementFactor, Execution, Expression, Factor e Symbol.

VI. Dificuldades na implementação e sugestão de melhorias

Considero como principal dificuldade a necessidade de adaptação à forma de pensar do Paradigma Orientado a Notificações. "Aceitar" que o funcionamento é diferente e esquecer os vícios dos outros paradigmas exige bastante esforço, especialmente para alguém que já trabalha há muitos anos com outros paradigmas. Mas a partir do momento que essa aceitação ocorre e os conceitos são entendidos a linguagem se torna intuitiva.

A forma de funcionamento do compilador é bem modelada, tendo todos os seus elementos separados no diretório "elements". Porém a classe principal NOPGraph poderia ser refatorada pois acabou se tornando uma *GodClass* [7] - uma única classe que controla toda a aplicação, de modo que as outras classes só mantém dados. E a implementação dos vetores só piorou esta situação, pois foi realizada dentro dela e não separadamente. Uma sugestão seria:

- mover os métodos de clonagem e instanciação de elementos dos vetores para uma classe própria
- criar uma classe utilitária para o método *checkFbeInstances* (e talvez outros checks*)
- o método *connectEntities* poderia ser quebrado em aproximadamente 7 ou 8 métodos e movido para a sua própria classe.

Também poderia ser adicionada a validação dos tipos dos atributos durante a fase de compilação para evitar problemas de tipagem nos geradores de código que utilizam esta informação.

A linguagem apresenta bom funcionamento e não tem limitações para a programação, sendo que novos elementos e evoluções sempre são bem-vindos. Outras linguagens que estão disponíveis há muito tempo continuam inovando e adicionando elementos, como por exemplo o Java que adicionou programação funcional a partir da versão 8 e tipagem dinâmica à partir da versão 10 [3]. Obviamente estas alterações quebram a compatibilidade com versões anteriores e exigem alterações na compilação e interpretação, além de adaptação e utilização por parte dos programadores.

VII. Considerações pessoais

A utilização do compilador da LingPON para a parte prática da disciplina a tornou bastante interessante. Como trabalhei com a parte da linguagem e precisei lidar com todas as fases da compilação, tive o privilégio de poder entender a fundo o funcionamento e construção de compiladores, da análise sintática à geração do código intermediário. E mesmo para os itens que não vi diretamente como a otimização e geração de código, tive contato nas apresentações dos trabalhos dos colegas de classe.

Particularmente achei essa disciplina muito importante e ela desmistificou algumas percepções que tinha sobre a compilação - creio que a mesma deveria estar incluída na graduação, provavelmente em uma forma mais simples mas de forma aos alunos entenderem a "mágica" que ocorre ao compilarem o código.

Por fim, uma sugestão para ampliar um pouco o conteúdo teórico da disciplina seria de abordar conceitos de linguagens interpretadas e compilação just-in-time (JIT) utilizada no Java e .Net.

VIII. Conclusão

O PON representa uma forma totalmente nova de se pensar e desenvolver software. Ao implementar uma linguagem própria (LingPON) é possível extrair o máximo do paradigma. Mas como todas as linguagens de programação, mudanças são bem-vindas. E com isto o suporte a vetores e regras de formação nesta linguagem se encaixou perfeitamente, extraindo ainda mais do seu potencial, reduzindo a repetição de código e permitindo uma utilização mais inteligente dos recursos da linguagem.

IX. Trabalhos Futuros

Conforme citado anteriormente seria interessante refatorar a classe NOPGraph e separar a parte de vetores que foi adicionada a ela. Em relação a especificação e utilização de vetores, pode-se validar a necessidade de suporte a utilização de atributos do tipo *Integer* como índices em qualquer lugar do código e a possibilidade de suportar matrizes multidimensionais.

References

[1] M. S. Pinho, "Programação C/C++". https://www.inf.pucrs.br/ pinho/LaproI/Vetores/Vetores.htm , visitada em 08/09/2018.

[2] Neto, João José (1987). Introdução à Compilação. Rio de Janeiro: LTC. 222 páginas. ISBN 978-85-216-0483-9.

[3] http://openjdk.java.net/projects/jdk/10/ , visitado em 15/09/2018

[3] RONSZCKA, Adriano Francisco. Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões. 2012. 236 f. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial) – Universidade Tecnológica Federal do Paraná, Curitiba, 2012.

[4] Roni Fábio Banaszewski. Paradigma Orientado a Notificações: Avanços e Comparações. Dissertação de Mestrado, CPGEI/UTFPR, Curitiba, 2009.

[5] Jean Marcelo Simão e Paulo Cézar Stadzisz. Inference Process Based on Notifications: The Kernel of a Holonic Inference Meta-Model Applied to Control Issues. IEEE Transactions on Systems, Man and Cybernetics. Part A, Systems and Humans, Vol. 39, Issue 1, 238-250, Digital Object Identifier 10.1109/TSMCA.2008.20066371, 2009.

[6] Jean Marcelo Simão, Roni Fábio Banaszewski, César Augusto Tacla e Paulo Cézar Stadzisz. Notification Oriented Paradigm (NOP) and Imperative Paradigm: A Comparative Study. Journal of Software Engineering and Applications (JSEA), 2012

[7] Riel, Arthur J. (1996). "Chapter 3: Topologies of Action-Oriented Vs. Object-Oriented Applications". Object-Oriented Design Heuristics. Boston, Massachusetts: Addison-Wesley. ISBN 0-201-63385-X. 3.2: Do not create god classes/objects in your system.