2
Conceitos Básicos

Neste capítulo são apresentados alguns conceitos e definições utilizados neste trabalho. Em particular, uma interpretação do termo middleware, uma definição de linguagens dinâmicas e a caracterização da linguagem Lua como tal. Ao final, também são apresentadas a definição de flexibilidade de software, que é utilizada ao longo deste texto, e uma descrição detalhada do CDN, que é utilizado como critério nas avaliações de flexibilidade.

2.1
Middleware

De forma geral, middleware designa uma infra-estrutura de software para construção de aplicações de um determinado tipo ou característica. Por exemplo, aplicações distribuídas tipicamente são construídas usando uma infra-estrutura de software presente nos nós que compõem a aplicação, denominado middleware de distribuição. Esse middleware oculta as diferenças entre as diversas plataformas de execução do ambiente distribuído (*e.g.* hardware, sistema operacional, linguagem de programação) ou mesmo a própria distribuição da aplicação.

O termo middleware é devido a esse tipo de software ser tipicamente visto como uma camada no meio da hierarquia de software de um sistema, ficando entre a aplicação e o software básico da plataforma. O middleware integra funcionalidades e recursos oferecidos por outros softwares sob um modelo de programação próprio, que consiste na forma com que essas funcionalidades são oferecidas para a aplicação. Isso se reflete não só nas interfaces de programação oferecidas (API - *Application Programming Interface*), mas também na forma com que o middleware espera ou impõe que a aplicação seja estruturada, que pode ser como um conjunto de objetos, tratadores de eventos, definições de políticas, etc.

Nesse sentido, podemos destacar os sistemas de middleware baseados em objetos distribuídos, denominados ORB (*Object Request Brokers*). Com esse tipo de middleware, a aplicação é construída como um conjunto de objetos, que podem estar espalhados pelo ambiente distribuído, mas que interagem

entre si através de chamadas de operações feitas por intermédio do ORB, que geralmente oculta a localização do objeto sendo chamado. Esse modelo de programação é uma adaptação para o domínio de aplicações orientadas a objeto do modelo de RPC (Birrell e Nelson, 1984) (*Remote Procedure Call*). Exemplos desse tipo de middleware incluem Java RMI (Grosso, 2001), CORBA (Object Management Group, 2002) e ICE (Henning, 2004).

Alternativas ao modelo de ORB incluem o modelo de *Publish-Subscribe*, onde a aplicação é construída como um conjunto de tratadores de eventos que são registrados para serem invocados quando um evento com determinadas características é recebido. Exemplos incluem Hermes (Pietzuch e Bacon, 2002) e Siena (Carzaniga et al., 2000).

Tanto a evolução natural das aplicações como o surgimento de novas aplicações mais complexas demandam que o middleware evolua similarmente para atender novas necessidades, como novos recursos ou modelos de programação diferentes ou mais abrangentes. Entretanto, o middleware idealmente não deve crescer em complexidade demasiadamente. Ao invés disso, é preferível que o middleware seja configurável e extensível (Eliassen et al., 1999). Como resultado dessa necessidade, surgiram os sistemas de middleware reflexivos e dinamicamente adaptáveis, que são aqueles que oferecem mecanismos para que o desenvolvedor ou a própria aplicação possa modificar sua implementação para se ajustar a diferentes necessidades e características. Nesse sentido, destacam-se as implementações de middleware baseadas em componentes de software reconfiguráveis ou reflexivos (Kon e Campbell, 1999; Coulson et al., 2002).

Uma das dificuldades do desenvolvimento e utilização de um middleware adaptável é a sua complexidade, em especial, quando a linguagem de programação não oferece recursos apropriados (Group, 2007). Como resultado disso, abordagens alternativas, que não são baseadas em middleware, têm ganhado notoriedade no desenvolvimento de software distribuído, como é o caso de AJAX (Garrett, 2005) e REST (Fielding, 2000). Em particular, essas tecnologias estão geralmente associadas a linguagens dinâmicas, que têm sido bastante utilizadas no desenvolvimento de aplicações distribuídas na Web. Entretanto, o uso de linguagens dinâmicas no desenvolvimento do middleware em si ainda é pouco disseminado (Vinoski, 2006).

2.2 Linguagens Dinâmicas

Não há uma definição específica do termo linguagem dinâmica na literatura. Neste trabalho, trataremos por linguagem dinâmica aquelas que ofereçam suporte direto para os seguintes recursos:

Interpretação é a capacidade de executar trechos de código da própria linguagem fornecidos em tempo de execução. O código interpretado executa integrado ao mesmo ambiente comum utilizado pelo resto do programa, compartilhando todas variáveis e funções globais por exemplo.

Tipagem dinâmica determina que cada valor tem um tipo associado em tempo de execução e cada tentativa de operação sobre esses valores resulta na verificação se o tipo permite a operação. Essas verificações permitem utilizar composições de código feitas durante a execução do programa de forma segura e controlada evitando erros críticos que possam terminar o programa inadvertidamente. Muitas linguagens dinâmicas evitam o uso de declarações explícitas de tipos e geralmente não realizam análises de código para detectar erros durante a compilação.

Reflexão computacional é o suporte da linguagem para analisar, em tempo de execução, as características do código e estruturas do próprio programa (*introspecção*) e interceder no comportamento desses elementos. Esse recurso permite que o programa faça verificações específicas sobre o próprio código ou mesmo sobre código sendo incorporado durante sua execução, de forma que também possa adaptá-lo adequadamente.

Gerência automática de memória é o suporte da linguagem de gerenciar automaticamente a alocação e liberação da memória utilizada com base em informações dinâmicas mantidas sobre o próprio programa. Esse recurso evita que o programador defina explicitamente quando novas porções de memória são requisitadas ou liberadas, o pode ser propenso a erros em muitos casos.

De forma geral, essas características facilitam a manipulação de um programa durante sua carga ou execução, de forma que ele possa se ajustar a diferentes usos. Em particular, algumas aplicações fazem uso dessa flexibilidade oferecida por linguagens dinâmicas para permitir que o usuário possa adicionar ou adaptar parte de suas funcionalidades (Boctor, 1999; Whitehead II et al., 2008). No caso de middleware, essa flexibilidade pode facilitar o ajuste do middleware a diferentes usos ou mesmo que o middleware possa se ajustar às necessidades futuras de uma única aplicação que é desenvolvida ao longo do tempo.

Por outro lado, linguagens dinâmicas introduzem uma sobrecarga devido à manutenção de informações para tipagem dinâmica e reflexão computacional. Outra crítica em relação a linguagens dinâmicas se refere à falta de verificações

estáticas do programa que garantam a ausência de certos tipos de erro já na compilação, mais especificamente, erros de tipagem.

2.2.1 A Linguagem Lua

A linguagem dinâmica utilizada nas implementações apresentadas neste trabalho é a linguagem Lua (Ierusalimschy, 2006). Lua é uma linguagem de propósito geral, projetada para ser simples porém expressiva e eficiente. Lua é uma linguagem imperativa com suporte para programação procedimental, que também oferece facilidades para descrição de dados através de estruturas de dados flexíveis denominadas *tabelas*.

As tabelas de Lua são arranjos associativos, isto é, um arranjo de valores em que cada um está associado a uma chave, que é distinta de todas as demais chaves naquela tabela e permite indexar o valor no arranjo. Qualquer valor em Lua pode ser utilizado como chave ou valor armazenado em uma tabela, inclusive as próprias tabelas. A única exceção é o valor `nil`, que é um valor especial utilizado genericamente para indicar a ausência de informações. Por exemplo, quando se indexa uma chave que não tem valor associado em uma tabela, o resultado é o valor `nil`. Tabelas permitem criar estruturas complexas para descrição de dados como seqüências, hierarquias e grafos em geral. Tabelas são criadas em Lua com a construção `{ ... }`, onde `...` indica os valores armazenados inicialmente na tabela usando a seguinte construção para cada valor `[<chave>] = <expressão>`. Opcionalmente a definição da chave pode ser omitida; nesse caso, assume-se que as chaves são números inteiros positivos consecutivos. A figura 2.1 ilustra a criação de tabelas em Lua usando formas sintáticas diferentes. O código apresentado à direita da figura é equivalente ao código da esquerda, porém utilizando açúcar sintático oferecido por Lua. As linhas 1–7 criam a variável local `sequence` com uma tabela contendo uma seqüência de cinco *strings*. As linhas 10–20 criam a variável local `struct` com uma tabela com informações estruturadas. Nesse caso, cada valor é armazenado com uma chave *string* que descreve o valor, denominada *campo*.

Lua também apresenta funcionalidades características de linguagens funcionais. Por exemplo, funções são valores de primeira-classe, ou seja, são manipulados de forma irrestrita como os demais valores da linguagem, tais como números, *strings*, tabelas, etc. Em particular, funções podem ser atribuídas a variáveis, parâmetros ou valores de retorno de uma chamada de função, assim como serem armazenadas em tabelas. Quando uma função é declarada, é criado um fecho, ou seja, uma instância da função ligada às variáveis que ela acessa no corpo do seu código, de forma que essas variáveis fiquem disponíveis

Figura 2.1: Estruturas de dados em Lua usando diferentes formas sintáticas.

```lua
local sequence = {
  [1] = "um",
  [2] = "dois",
  [3] = "tres",
  [4] = "quatro",
  [5] = "cinco",
}
print(sequence[4]) --> quatro

local struct = {
  ["name"] = "Fulano",
  ["age"] = 35,
  ["address"] = {
    ["street"] = "Rua sem saida",
    ["number"] = "S/N",
    ["postalcode"] = 22222000,
    ["city"] = "Rio de Janeiro",
    ["country"] = "Brasil",
  },
}
struct["name"] = "Beltrano"
struct["address"]["city"] = "RJ"
```

```lua
local sequence = {
  "um",
  "dois",
  "tres",
  "quatro",
  "cinco",
}
print(sequence[4]) --> quatro

local struct = {
  name = "Fulano",
  age = 35,
  address = {
    street = "Rua sem saida",
    number = "S/N",
    postalcode = 22222000,
    city = "Rio de Janeiro",
    country = "Brasil",
  },
}
struct.name = "Beltrano"
struct.address.city = "RJ"
```

na execução da função. A figura 2.2 ilustra a criação de uma função fatorial recursiva que utiliza a variável `fatorial` do seu fecho, onde a própria função é armazenada, para chamá-la recursivamente (linha 6).

Figura 2.2: Fecho de função em Lua usando diferentes formas sintáticas.

```lua
local fatorial
fatorial = function(n)
  if n <= 1 then
    return 1
  else
    return n * fatorial(n-1)
  end
end

print(fatorial(5)) --> 120
```

```lua
local function fatorial(n)
  if n <= 1 then
    return 1
  else
    return n * fatorial(n-1)
  end
end

print(fatorial(5)) --> 120
```

A possibilidade de armazenar funções em tabelas permite criar programas orientados a objetos usando tabelas como objetos. Objetos podem ser criados como tabelas que definem campos contendo valores dos seus atributos e funções que implementam suas operações. Em particular, as funções que implementam operações recebem um parâmetro adicional denominado `self`, que informa o objeto sobre o qual a operação é executada. A figura 2.3 ilustra a criação de um objeto usando uma tabela.

Lua é uma linguagem dinâmica, pois apresenta as três características vistas anteriormente. Apesar de Lua utilizar uma etapa de pré-compilação do código antes de ser efetivamente interpretado pela máquina virtual, ela permite executar código-fonte obtido durante a execução dos programas, pois o compilador é parte integrante do ambiente de execução de Lua. Esse recurso é disponibilizado através de algumas funções oferecidas pela biblioteca padrão

Figura 2.3: Objeto em Lua usando diferentes formas sintáticas.

```lua
local person = {
  ["name"] = "Fulano",
  ["age"] = 35,
}
person["show"] = function(self)
  print(self["name"], self["age"])
end

person["show"](person)
```

```lua
local person = {
  name = "Fulano",
  age = 35,
}
function person:show()
  print(self.name, self.age)
end

person:show() --> Fulano    35
```

de Lua, como por exemplo a função `loadstring`, que cria uma função cujo corpo é definido pelo código fornecido em uma *string*. A figura 2.4 ilustra o uso dessa função para interpretar um código que cria uma tabela com valores de fatoriais de números inteiros e depois imprime esses valores. Um recurso similar é utilizado para carregar bibliotecas de extensão que são escritas em Lua através da função `require`.

Figura 2.4: Interpretação de código-fonte em um programa Lua.

```lua
local code = [[
  local fat_table = {}
  for i = 1, 5 do
    fat_table[i] = fatorial(i)
  end
  return fat_table
]]

local fats = loadstring(code)() -- interpreta o codigo em 'code'
for i = 1, #fats do
  print("Fatorial de "..i..": "..fats[i])
end
```

Lua é dinamicamente tipada e não oferece mecanismos de verificação estática de tipo. Todas as variáveis, parâmetros ou valores de retorno de funções, chaves ou valores de tabela podem assumir valores de diferentes tipos de forma irrestrita durante a execução do programa. Além disso, as funções podem receber um número arbitrário de argumentos independente dos parâmetros formais definidos na declaração da função. Quando o número de argumentos passados em uma chamada for menor que os parâmetros formais da função, os últimos parâmetros assumem o valor `nil`. Se o número de parâmetros for maior, os valores adicionais são descartados. O mesmo acontece com os valores de retorno. Essa flexibilidade dá mais liberdade ao programador, porém aumenta a possibilidade de erros e equívocos que não são detectados na compilação ou carga do código.

Lua também oferece mecanismos de reflexão computacional. Os mecanismos de introspecção permitem identificar os tipos de valores (função `type`), a estrutura das tabelas (função `next`), entre outros aspectos do programa. Além disso, apesar da biblioteca de suporte a depuração de Lua não ser projetada

como um recurso de programação usual, ela oferece mecanismos de introspecção como a inspeção das variáveis de um fecho (função `debug.getupval`). Contudo, muitos outros aspectos de Lua não podem ser inspecionados de forma trivial, como é o caso dos parâmetros formais ou o código de uma função. Os mecanismos de intercessão só permitem interceder em alguns comportamentos, em especial aqueles que normalmente resultariam em erros, como a aplicação de operações em valores cujo o tipo não define aquela operação. Um dos principais mecanismos de intercessão de Lua possibilita redefinir funções globais do programa, inclusive funções essenciais da biblioteca padrão, como a `loadstring` ilustrada na figura 2.4, o que permite introduzir mecanismos como pré-processamento de código a ser executado.

Outro mecanismo importante de intercessão em Lua são as *meta-tabelas*. Meta-tabelas são tabelas comuns associadas a valores, que podem conter campos especiais que definem parte do comportamento do valor associado, denominados *meta-métodos*. Por exemplo, o campo `__index` pode ser uma função que define o comportamento do valor quando este é indexado como uma tabela. Entretanto, há algumas limitações no uso de meta-tabelas. Por exemplo, no caso de tabelas, o campo `__index` só é utilizado quando a tabela não possui a chave sendo indexada, caso contrário esse mecanismo de intercessão é ignorado. O mesmo é válido para o campo `__newindex` que define o comportamento quando um novo valor é associado a uma chave em uma tabela. Outra limitação é que funções, assim como números e *strings*, compartilham uma única meta-tabela, não sendo possível redefinir o comportamento individual de uma função como é possível com tabelas[1].

Apesar dessas limitações, algumas técnicas podem ser utilizadas para ampliar a aplicabilidade dos mecanismos de intercessão de Lua. Um exemplo disso é substituir funções e tabelas por representantes ou *proxies*, que são objetos especiais que usam os mecanismos de intercessão de Lua para delegar operações às funções ou tabelas que eles representam. Dessa forma, todas as operações feitas através dos *proxies* podem ser redefinidas, bastando alterar a implementação dos mecanismos de intercessão utilizados para delegar essas operações. Por exemplo, a figura 2.5 ilustra a implementação de uma função que cria um *proxy* através de uma tabela vazia cuja meta-tabela define que toda operação feita no *proxy* é delegada a outro valor (linha 1). Essa função é então utilizada para criar um *proxy* para uma função que calcula o fatorial de um número (linha 19) e outro *proxy* para uma tabela (linha 27). Os *proxies* são utilizados no lugar da função e da tabela originais (linha 29). Entretanto, agora

[1]Essa limitação pode ser contornada com o uso de ambientes de função, que são tabelas que podem ser associadas individualmente a funções.

é possível modificar a forma com que a função é chamada individualmente, assim como a forma com que a tabela é indexada independentemente das chaves já armazenadas. Para tanto, basta modificar a implementação dos meta-métodos de cada *proxy* da forma adequada, como é feito pelas funções das linhas 35, 43 e 49, que passam a imprimir mensagens de acompanhamento em cada um desses esses eventos.

Figura 2.5: Uso de meta-tabelas para implementação de *proxies* de valores.

```
1  local function newproxy(value)
2    local proxy = {}
3    setmetatable(proxy, {
4      __index=function(self, k)
5        return value[k]
6      end,
7      __newindex=function(self, k, v)
8        value[k] = v
9      end,
10     __call=function(self, ...)
11       return value(...)
12     end,
13     -- outros meta-metodos
14   })
15   return proxy
16 end
17
18 local fat
19 fat = newproxy(function(n)
20   if n <= 1 then
21     return 1
22   else
23     return n * fat(n-1)
24   end
25 end)
26
27 local fats = newproxy({})
28 for i = 1, 5 do
29   fats[i] = fat(i)
30   print(i.."! = "..fats[i])
31 end
32 local funcMT = getmetatable(fat)
33
34 local old = funcMT.__call
35 function funcMT:__call(...)
36   print("PROFILING: call to",self)
37   return old(self, ...)
38 end
39
40 local tabMT = getmetatable(fats)
41
42 local old = tabMT.__index
43 function tabMT:__index(key)
44   print("PROFILING: index",self)
45   return old(self, key)
46 end
47
48 local old = tabMT.__newindex
49 function tabMT:__newindex(key,val)
50   print("PROFILING: change",self)
51   return old(self, key, val)
52 end
53
54 for i = 1, 5 do
55   fats[i] = fat(fats[i])
56   print(i.."!! = "..fats[i])
57 end
```

O uso de *proxies* é custoso, tanto em relação ao desempenho, pois implica em uma sobrecarga de memória e processamento, quanto em relação à facilidade de uso, pois introduz uma complicação maior na criação dos valores, além de impor algumas limitações devido às próprias limitações dos mecanismos de intercessão de Lua, resultando em uma maior propensão a erros. Portanto, o uso dessa abordagem deve ser restrita a pontos estratégicos do programa. Por exemplo, uma abordagem similar pode ser utilizada em programas Lua orientados a objetos para permitir interceder nas chamadas de operações dos objetos. Para tanto, cada objeto é associado a uma meta-tabela que representa a classe do objeto. Os mecanismos de intercessão da meta-tabela são utilizados para fazer com que as operações definidas pela classe sejam oferecidas pelos seus objetos, como ilustrado no exemplo da figura 2.6. Neste exemplo, a tabela `QueueMembers` (linha 1) é usada pelo meta-método `__index` definido em `QueueMetatable` (linha 21) para obter os membros oferecidos pelos

objetos da classe representada por essa meta-tabela. A função `Queue` é utilizada para criar objetos dessa classe (linha 25).

Figura 2.6: Uso de meta-tabelas para implementação de classes de objeto.

```
1  local QueueMembers = {
2    -- atributos default
3    head = 0,
4    tail = 0,
5    -- operacoes oferecidas
6    empty = function(self)
7      return self.head >= self.tail
8    end,
9    enqueue = function(self, value)
10     self.tail = self.tail + 1
11     self[self.tail] = value
12   end,
13   dequeue = function(self)
14     if self.head < self.tail then
15       self.head = self.head + 1
16       return self[self.head]
17     end
18   end,
19 }
20 local QueueMetatable = {
21   __index = function(self, name)
22     return QueueMembers[name]
23   end,
24 })
25 local function Queue()
26   local queue = {}
27   setmetatable(queue, QueueMetatable)
28   return queue
29 end
30
31 local queue = Queue()
32 queue:enqueue("first")
33 queue:enqueue("second")
34 queue:enqueue("third")
35 while not queue:empty() do
36   print(queue:dequeue())
37 end
```

Essa abordagem utiliza os mecanismos de intercessão de Lua para permitir estender os objetos com operações definidas pela sua classe. Dessa forma, é possível interceder nas operações oferecidas pelos objetos mudando a implementação do campo `__index` da meta-tabela `QueueMetatable`. Apesar dessa abordagem estar restrita a intercessão em operações fornecidas pela classe, é um mecanismo útil em aplicações orientadas a objeto, onde as ações são feitas através de operações de objeto em sua grande parte. Uma abordagem similar é utilizada para introduzir recursos de programação orientada a aspectos em Lua (Batista e Vieira, 2007).

Lua também oferece suporte ao conceito de co-rotinas. Uma co-rotina representa uma linha de execução independente em Lua. Após a sua criação, a execução da co-rotina pode ser iniciada através da função `coroutine.resume` de Lua. Nesse caso, a co-rotina passa a executar até uma chamada da função `coroutine.yield`, que faz com que sua execução seja suspensa. Nesse momento, a chamada original de `coroutine.resume` retorna. O contexto de execução da co-rotina suspensa permanece inalterado até que uma nova chamada de `coroutine.resume` seja feita naquela co-rotina, o que fará com que sua execução seja restaurada e ela prossiga imediatamente após a chamada da operação `coroutine.yield`. Esse modelo de co-rotinas baseado em duas funções, uma para restaurar a execução e outra suspendê-la, é denominado *co-rotinas assimétricas* e pode ser utilizado em diferentes usos, em particular para implementar suporte a concorrência como será visto na seção 3.3.4.

Assim como outras linguagens dinâmicas, Lua ganhou maior popularidade recentemente e alguns de seus usuários clamam que suas características

permitem o desenvolvimento de aplicações eficientes e flexíveis de forma mais fácil e produtiva (Hirschi, 2007). Entretanto, poucos estudos têm sido feitos para validar essas asserções de forma mais objetiva como é proposto em (Eden e Mens, 2006).

2.3 Flexibilidade de Software

Segundo o glossário de terminologia de Engenharia de Software da IEEE (IEEE Standard Board, 1991), a flexibilidade de software se refere à facilidade com que o software pode ser modificado para se ajustar ao uso em aplicações ou ambientes diferentes daquele para o qual foi especificamente projetado.

> ***Flexibility****: The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.*

Neste trabalho, utilizaremos uma definição um pouco mais abrangente e comumente aceita que leva em consideração também eventuais correções do software ou mesmo ajustes para usos futuros que não tenham sido especificamente previstos no seu projeto. Com base nisso, distinguimos três tipos de flexibilidade, conforme descritas abaixo:

Portabilidade Facilidade de adaptar o software para um mesmo uso, mas em diferentes plataformas e ambientes de execução. Um exemplo é a capacidade de utilizar uma mesma implementação de middleware com as mesmas funcionalidades tanto no desenvolvimento de aplicações em um computador pessoal quanto em um celular. Esse tipo de flexibilidade está diretamente relacionada às dependências do software e não ao próprio software em si, ou seja se a infra-estrutura de execução exigida pelo software pode ser fornecida em diferentes plataformas de computação.

Configurabilidade Facilidade de adaptar o software a um uso específico previsto ou esperado. Um exemplo seria habilitar o recurso de interceptação de chamadas em um ORB para ser utilizado em aplicações que precisem desse tipo de funcionalidade. Esse tipo de flexibilidade é especificamente projetada no software e está diretamente ligada à abrangência das suas aplicações. A configurabilidade pode se referir tanto a facilidade de adaptar o software durante sua execução como durante o desenvolvimento de uma nova versão que substituirá uma anterior.

Manutenibilidade Facilidade de corrigir inadequações do software que não tenham sido identificadas originalmente. Um exemplo seria a correção de um erro de programação na sua implementação ou a introdução de novo recurso não previsto originalmente no seu projeto. Esse tipo de flexibilidade está diretamente ligada a evolução do software e também engloba tanto alterações durante a execução do software como no desenvolvimento de uma nova versão.

No caso específico de middleware, os três tipos de flexibilidade são importantes. A portabilidade é importante para ampliar sua utilização no desenvolvimento de diferentes aplicações. A configurabilidade e a manutenabilidade são importantes como forma de não comprometer a manutenção das aplicações construídas sobre o middleware e também facilitar a evolução do próprio middleware em virtude da descoberta de novos usos potenciais.

2.3.1 Avaliação de Flexibilidade

De forma geral, flexibilidade não pode ser vista como uma característica absoluta, pois a facilidade de modificação tipicamente depende diretamente de uma situação de uso específica sendo considerada (Eden e Mens, 2006). Por exemplo, um editor de texto pode ser facilmente ajustável para permitir a definição de novos comandos, porém pode ser difícil permitir ler novos formatos de arquivo. Consequentemente, é necessário definir casos ou situações de uso que servirão de base para a avaliação da flexibilidade. O capítulo 4 apresenta os casos de alteração utilizados nas avaliações feitas nesse trabalho.

Com a identificação de um conjunto de casos de modificação relevantes, é necessário definir como avaliar a facilidade da modificação. Neste trabalho, foram consideradas duas alternativas para avaliação dessa facilidade. A primeira alternativa consiste em utilizar métricas quantitativas sobre o custo absoluto da introdução de uma dada alteração, como por exemplo o número de linhas de código (LOC), funções ou classes de objeto modificadas (Eden e Mens, 2006). Entretanto, esse tipo de avaliação, apesar de bem objetivo, desconsidera aspectos importantes relacionados à facilidade de entender e planejar corretamente as modificações, atividades estas que envolvem questões subjetivas relacionadas a aspectos cognitivos. Por isso, a alternativa adotada se baseia em análises mais subjetivas, mas que também levam em consideração questões relacionadas à facilidade de uso. Para tanto, são utilizadas as Dimensões Cognitivas de Notações, que é um *framework* utilizado na avaliação da iteração humano-computador (IHC), como descrito na seção 2.4.

Nessas avaliações de flexibilidade, também é importante fazer a distinção entre dois tipos de atividade de modificação do software descritos abaixo. Essa distinção é importante pois tipicamente implicam em necessidades distintas. Essa classificação é baseada em uma classificação proposta em (Green e Blackwell, 1998).

Definitiva Atividade de introduzir uma modificação necessária que deve ser efetivamente incorporada ao software. Tipicamente, essa atividade envolve um planejamento mais cuidadoso e tem suas conseqüências bem definidas. O foco principal é evitar erros e possíveis problemas introduzidos pela modificação e garantir a qualidade do software resultante.

Exploratória Atividade de introduzir uma modificação com propósito de investigação, mas que possivelmente será descartada ou exigirá uma atividade adicional (definitiva) para ser incorporada efetivamente. Tipicamente essa atividade envolve uma avaliação de modificações no software, inclusive a identificação dos erros e problemas introduzidos. O foco está centrado no aprendizado do software e suas características e não no software resultante.

No caso específico de middleware, pode-se destacar duas atividades principais de modificação consideradas neste trabalho: (a) o desenvolvimento de produtos de middleware para construção de aplicações; (b) a pesquisa e ensino sobre desenvolvimento de middleware inovadores. No primeiro caso, as modificações são geralmente definitivas visando a evolução ou melhoria de um produto de middleware, baseando-se em implementações validadas e visando um produto final de qualidade. No segundo caso, as modificações são muitas vezes exploratórias visando a validação ou compreensão de uma técnica de implementação de middleware. Neste caso, a implementação sendo alterada deve ser acessível aos pesquisadores e estudantes. Neste trabalho, avaliaremos a flexibilidade do middleware levando em consideração esses dois tipos de atividades.

2.4 Dimensões Cognitivas de Notações

Medir a flexibilidade de um software é uma tarefa complexa, pois a avaliação objetiva da facilidade de realizar uma tarefa, como a modificação de software, envolve diversos conceitos subjetivos, que são difíceis de ser quantificados. Em (Green, 1989), foi proposto o *framework* de *Dimensões Cognitivas de Notações* (CDN, do inglês *Cognitive Dimensions of Notations*), que consiste em um conjunto de critérios ou dimensões que influenciam

a facilidade de um ser humano em manipular artefatos descritos em uma notação arbitrária. O objetivo principal do CDN foi definir um vocabulário ou um conjunto básico de questões de facilidade de uso que possam ser utilizados por desenvolvedores de software em geral, sem a necessidade de treinamento prolongado ou conhecimento multi-disciplinar (Blackwell et al., 2001). Desde então, o CDN já foi utilizado na avaliação de linguagens de programação (Clarke, 2001), ambientes de programação visual (Green e Petre, 1996), APIs (Clarke, 2006), etc. A facilidade de apresentação do CDN e sua aplicação no domínio de desenvolvimento de software foram decisivas para sua adoção neste trabalho como critério de avaliação de facilidade de modificação, ou seja, flexibilidade.

A definição de notação do CDN é bastante genérica, pois considera qualquer sistema com que um ser humano pode interagir fornecendo e obtendo informações relativas a um artefato sendo manipulado. Além disso, a notação pode incluir um conjunto de sub-dispositivos que auxiliam as atividades do usuário, tanto produzindo informações como modificando a própria notação. Em geral, as avaliações de software usando o CDN consideram a interface do software com seu usuário como parte da notação sendo avaliada e o resultado da utilização do software como o artefato sendo produzido. Nesse sentido, o OiL pode ser visto como uma notação cujo artefato produzido ou descrito é uma implementação de middleware. Nesse caso, o OiL compreende, entre outras coisas, a linguagem Lua, a forma com que sua implementação é organizada e o código disponível na sua implementação que é reutilizado no desenvolvimento do middleware. Os sub-dispositivos incluem editores de texto, ambientes e ferramentas de programação, interpretadores, etc. No contexto deste trabalho, os sub-dispositivos do OiL não são considerados pois estão mais relacionados às ferramentas e mecanismos de auxílio a programação em linguagens dinâmicas, cuja avaliação foge do escopo deste trabalho.

O conjunto de dimensões do CDN não é definitivo ou completo, uma vez que não há garantia que as dimensões cubram todos os aspectos relevantes da facilidade de uso. Atualmente 14 dimensões são consideradas estabelecidas e algumas outras, chamadas dimensões candidatas, estão sendo consideradas para inclusão neste conjunto (Blackwell et al., 2001). Nem todas as dimensões do CDN são ortogonais, ou seja, muitas tratam de questões que se sobrepõem ou se assemelham. Além disso, algumas dimensões podem ser irrelevantes em certas avaliações. Neste trabalho, as dimensões do CDN são utilizadas como critério de avaliação da facilidade de modificação do OiL. O objetivo dessa avaliação é identificar as características particulares da sua implementação, com foco especial para as características dinâmicas da linguagem Lua, que

facilitam sua modificação, ou seja, aumentam sua flexibilidade. A seguir são apresentadas as 14 dimensões estabelecidas do CDN.

2.4.1 Viscosidade

A viscosidade se caracteriza pela resistência a mudanças. Quanto mais custoso é introduzir uma alteração, mais viscosa é a notação. Tomando como exemplo a linguagem Java, a introdução de uma nova operação em uma interface, que é implementada por diversas classes que estendem classes diferentes, exige que essa operação seja implementada individualmente em cada uma dessas classes. Portanto, em um cenário em que seja necessário incrementar essas interfaces com freqüência, o sistema é considerado muito viscoso. De forma geral, a viscosidade de uma implementação de software pode ser reduzida através da estruturação da implementação de forma a acomodar certas mudanças mais facilmente. No exemplo citado anteriormente, a restruturação da hierarquia de classes pode permitir a utilização do mecanismo de herança para introduzir implementações de novas operações através da modificação de uma ou poucas classes.

A viscosidade é uma das dimensões mais diretamente relacionadas a flexibilidade de software, pois ela caracteriza o custo efetivo da realização de uma mudança. Por isso, de um modo geral uma alta viscosidade compromete diretamente a flexibilidade tanto em mudanças definitivas como exploratórias. Entretanto, a viscosidade não leva em consideração as dificuldades relacionadas ao entendimento ou planejamento da mudança. Apesar de comprometer a flexibilidade geral do sistema, a alta viscosidade pode ser útil como forma de proteger partes críticas do sistema, ou mais propensas a erros, de modificações inadvertidas. Contudo, do ponto de vista da flexibilidade, é preferível evitar a propensão a erros (outra dimensão do CDN) no sistema de forma geral.

De um modo geral, as características de linguagens dinâmicas reduzem a viscosidade do software durante a sua execução, pois permitem modificações que só são possíveis durante o desenvolvimento do programa em linguagens não-dinâmicas. Por exemplo, a interpretação de código facilita que o usuário do software incorpore novo código que estende ou substitui parte da implementação. Um exemplo disso é a utilização de linguagens de *script* embutidas que permitem alterar ou estender a aplicação através de *scripts* do usuário. A reflexão computacional também facilita a introdução de modificações automáticas pré-definidas no software. A gerência automática de memória evita que o desenvolvedor tenha de ajustar o modelo de liberação de memória em virtude de eventuais modificações no ciclo de vida de componentes do software.

Quando comparadas a linguagens que se baseiam em declarações explícitas de tipos, linguagens dinâmicas tender a ser menos viscosas. Isso é devido principalmente à necessidade de manter as declarações explícitas consistentes em todas as modificações. Algumas linguagens diminuem esse problema através do uso de abordagens de verificação mais flexíveis, como inferência de tipos (Milner, 1978).

2.4.2 Compromissos Prematuros

Essa dimensão se caracteriza por restrições que o usuário deve impor sobre suas futuras necessidades em um momento em que elas não podem ser determinadas apropriadamente. Um exemplo são os *arrays* estáticos de C, cujo tamanho deve ser definido antes da execução da aplicação, porém em muitas situações o tamanho realmente necessário só pode ser definido precisamente quando a aplicação estiver executando.

Independente do custo da correção de compromissos prematuros inadequados (viscosidade), a necessidade de definí-los dificulta o uso da notação, pois faz com que o usuário se preocupe em minimizar a possibilidade de definir um compromisso de forma inadequada. Portanto, apesar de um compromisso prematuro não implicar necessariamente em um custo mecânico da modificação, eles implicam em um custo mental para o usuário. É problemática a associação de um alto índice de compromissos prematuros com alta viscosidade, pois os ajustes dos compromissos assumidos se tornam custosos. Adicionalmente, compromissos prematuros são problemáticos em modificações definitivas pelo possível impacto de compromissos inadequados na qualidade do produto final. Em modificações exploratórias, os compromissos prematuros são menos problemáticos, mas podem distrair o desenvolvedor do aprendizado relativo ao experimento.

As características das linguagens dinâmicas podem efetivamente evitar a necessidade de compromissos prematuros, especialmente aqueles que só podem ser assumidos corretamente com base em informações de tempo de execução. Isso é comum no caso de estruturas de dados que associam informações utilizadas por componentes externos com elementos mantidos pelo sistema. Por exemplo, em um sistema de controle de uma grade computacional, é importante que cada nó da grade controlado pelo sistema possa ter um conjunto de informações associadas que são utilizadas para selecionar o nó mais adequado para uma dada computação. Esse conjunto de informações pode ser difícil de ser previsto no desenvolvimento do sistema, sem conhecer as características dos nós ou as necessidades dos usuários e outras aplicações

que possam vir a interagir com o sistema depois que ele estiver em execução. Em geral problemas desse tipo são solucionados em linguagens não-dinâmicas implementando estruturas dinâmicas que possam armazenar informações de diferentes tipos, e também possam ser inspecionadas e modificadas. Já em linguagens dinâmicas, a tipagem dinâmica facilita a implementação dessas estruturas e a reflexão computacional oferece suporte direto para inspeção dessas estruturas. Além disso, a reflexão computacional também ajuda a evitar compromissos prematuros em outros aspectos do sistema além de estruturas de dados, pois permite implementar ajustes do comportamento e da própria estrutura interna do sistema.

2.4.3 Dependências Ocultas

Essa dimensão se caracteriza pela quantidade de dependências importantes entre as partes do sistema que não estejam diretamente aparentes para o usuário. Dependências ocultas impedem um bom entendimento do sistema, como por exemplo identificar a importância ou a necessidade de cada parte individual do sistema. Um exemplo é a revisão de um programa extenso por todas as chamadas de uma função disponível globalmente cujo comportamento deve ser alterado.

Dependências ocultas dificultam identificar e compreender as implicações de modificações, independentemente da dificuldade de introduzí-las (viscosidade) ou da sua corretude ou adequação (propensão a erros). Excetuando as modificações que claramente causam pouco impacto na implementação existente, as dependências ocultas geralmente dificultam o entendimento e planejamento da alteração. Em modificações exploratórias, onde o foco está no aprendizado, as dependências ocultas são particularmente problemáticas.

Em geral, declarações de tipos permitem explicitar algumas dependências. Por exemplo, o nome do tipo na declaração de variáveis e parâmetros serve de indicação dos pontos onde os valores daquele tipo são utilizados, o que é particularmente interessante para tipos definidos pelo programador. Inclusive, linguagens fortemente tipadas permitem explicitar todos os relacionamentos importantes do programa através do sistema de tipos. Declarações de tipo também podem ser utilizadas por ferramentas auxiliares que facilitem a visualização de dependências importantes. Por outro lado, as próprias declarações de tipo podem introduzir novas dependências ocultas. Por exemplo, em Java, não é fácil identificar as implicações da modificação de uma interface que é herdada por uma variedade de classes e outras interfaces. Por essas razões, a tipagem estática geralmente implica em um menor número

de dependências ocultas que ficam explicitas no código ou na compilação do programa.

2.4.4 Provisão

A provisão se caracteriza pelo grau de impacto e relevância que as decisões tomadas durante a criação do produto afetam o resultado final. Uma alta provisão permite adiar decisões (compromissos prematuros) de forma que possam ser definidas mais adequadamente em um momento posterior à medida que o produto final vai tomando mais forma. Na avaliação de notações, a relevância dessa dimensão se resume à possibilidade de produzir rascunhos ou moldes como passo intermediário para o produto final, reduzindo o impacto de compromissos prematuros inadequados. A provisão também pode ser vista como um mecanismo para reduzir a viscosidade em relação especificamente a compromissos prematuros.

De uma forma geral, a provisão melhora a flexibilidade pois permite realizar mais facilmente modificações que não são completamente compreendidas ou planejadas antes da sua realização. Isso é especialmente importante em modificações experimentais, pois comumente envolvem um aprendizado durante o desenvolvimento que pode ser necessário para tornar mais claras as decisões adequadas que devem ser tomadas.

A tipagem dinâmica, assim como a inferência de tipos na tipagem estática, funcionam como mecanismos de provisão, pois permitem que algumas porções do programa se ajustem automaticamente às alterações nos tipos dos valores feitas durante o desenvolvimento. A reflexão computacional pode ser vista como um mecanismo de provisão, pois o programador pode utilizar os mecanismos de intercessão oferecidos para alterar alguns comportamentos e com isso contornar algumas decisões. Como exemplo disso, considere que seja implementado em Lua o padrão *Observer* (Gamma et al., 1994). O programador então deve decidir a interface dos observadores, que podem ser funções, objetos com uma operação específica ou co-rotinas. Contudo, os mecanismos de reflexão de Lua permitem alterar o comportamento de qualquer um desses valores de forma a simular os demais. Mecanismos de provisão similares podem ser oferecidos em linguagens estáticas através de abstrações como macros. Entretanto, elas não estão disponíveis em tempo de execução.

2.4.5 Avaliação Progressiva

Essa dimensão se caracteriza pela possibilidade de verificar ou avaliar o produto ainda parcialmente confeccionado. No caso de programas, isso se reflete na possibilidade de compilá-lo ou executá-lo antes de estar completamente terminado ou correto. Um exemplo seria poder avaliar o funcionamento do mecanismo de realização de chamadas remotas de um ORB, usando como parâmetros, valores cujos tipos já têm uma codificação para formato de transmissão implementada, mesmo que nem todos os tipos da linguagem já tenham sua codificação implementada.

Avaliações progressivas melhoram a flexibilidade pois facilitam a modificação de programas complexos através de validações incrementais, durante todo o desenvolvimento. Em modificações definitivas, essa facilidade permite utilizar mecanismos de garantia de qualidade em momentos mais iniciais do desenvolvimento, diluindo a dificuldade dessa tarefa durante todo o processo. Em modificações exploratórias, esse recurso é ainda mais importante, pois permite ao programador realizar experimentações mais facilmente e com isso confrontar mais informações no seu aprendizado.

A tipagem estática dificulta avaliações progressivas, pois o programa precisa estar correto integralmente, mesmo que o programador só esteja interessado em avaliar uma parte do programa. Isso constitui uma das grandes vantagens da tipagem dinâmica, uma vez que não é necessário verificar o programa por completo antes de sua execução, pois essas verificações são feitas durante a execução do programa e apenas nas partes sendo executadas. O suporte a interpretação também facilita avaliações progressivas, pois evita a necessidade do passo de compilação. Em particular, o uso de consoles interativos para programação permitem a construção de programas de forma incremental e interativa.

2.4.6 Propensão a Erros

Essa dimensão se caracteriza pela quantidade de enganos que um usuário é levado a cometer por conta da notação, em particular aqueles que não são facilmente detectados ou são detectados inadequadamente, por exemplo, em um momento muito tardio, quando a correção é muito custosa. Um exemplo disso é o uso de aritmética de ponteiros em C, que permite acessos a áreas de memória inválidas que não podem ser detectadas facilmente pelo compilador. Esse é um tipo de erro comum dos programadores.

A propensão a erros é extremamente danosa para a flexibilidade, pois torna as modificações mais difíceis, uma vez que o programador deve ter mais cuidado com o planejamento da modificação. Essa dimensão é mais problemática quando combinada com a dificuldade de corrigir os erros cometidos (viscosidade). Em modificações exploratórias, a propensão a erros dificulta o aprendizado pois os erros ou a preocupação em evitá-los distrai o programador do objetivo principal. Em atividades de modificação definitiva, a propensão a erros é ainda mais problemática pois compromete a garantia de qualidade do resultado.

A tipagem dinâmica tem por característica identificar erros durante a execução do programa, quando a correção pode ser impossível em algumas aplicações. Por essa razão, linguagens dinâmicas são mais propensas a erros, inclusive aqueles que são comumente cometidos por programadores, como erros de digitação. Por outro lado, em linguagens que se baseiam em declarações explícitas de tipo para realizar verificações, o programador pode cometer erros nas próprias declarações que, apesar de geralmente serem facilmente identificados, podem não ser corrigidos facilmente (viscosidade). Por exemplo, em Java, é comum utilizar classes para definir novos tipos como no exemplo da figura 2.7, em que a classe `Hello` define um tipo que permite a operação `sayHello` (linha 1). A classe `RemoteHello` é sub-classe de `Remote` que define uma implementação alternativa do tipo `Hello` (linha 4). Contudo, Java não permite uma classe implementar o tipo definido por outra classe não relacionada. Apesar do compilador facilmente identificar o erro, sua correção pode ser custosa, pois implica em dividir a classe `Hello` em duas partes: uma interface e uma classe, cada uma com um nome distinto. Em seguida, o programador deve rever todas as referências (dependências ocultas) que a utilizam como implementação (linha 12) ou como tipo (linha 10).

Figura 2.7: Definição de tipo como uma classe Java.

```
1  class Hello {
2    String sayHello() { return "Hello"; }
3  }
4  class RemoteHello extends Remote implements Hello {
5    String sayHello() { /* ... */ }
6  }
7
8  public Program {
9    public static void main(String[] args) {
10     Hello hello;
11     if (args.length == 0)
12       hello = new Hello();
13     else
14       hello = new RemoteHello(args[0]);
15     /* ... */
16   }
17 }
```

Muitas das verificações de tipo estáticas não validam todos os aspectos do programa, e a verificação formal da corretude de programas é uma tarefa complexa e custosa. Como alternativa, alguns desenvolvedores adotam abordagens mais pragmáticas para verificação da corretude de aplicações. Um exemplo são testes de regressão (Myers, 1979), que são especificamente elaborados para identificar erros mais relevantes de um programa. Essa abordagem de verificação é comumente utilizada para detectar erros em programas escritos em linguagens dinâmicas, inclusive os erros de tipo.

Por outro lado, mecanismos de gerência automática de memória, como a coleta automática de lixo de Lua, efetivamente diminuem a propensão a erros como a utilização de memória não alocada ou a falta de liberação de porções de memória que não são mais utilizadas (vazamento de memória).

2.4.7 Mecanismos de Abstração

Essa dimensão se refere à utilização de mecanismos de abstração, que possui três variações:

Barreira de abstração se refere ao número de abstrações que são necessárias para utilização da notação. Como exemplo, para escrever qualquer programa em Java (*e.g.* imprimir uma mensagem na tela), é necessário conhecer pelo menos os conceitos como classes, objetos, métodos, etc. Esses conceitos constituem a barreira de abstração da linguagem. Uma grande barreira de abstração dificulta o aprendizado da notação e o seu uso também, pois o usuário deve lembrar de todas as abstrações.

Demanda de abstrações se refere à necessidade do usuário em definir novas abstrações para utilizar a notação, mesmo que essas novas abstrações não sejam adequadas para a descrição do problema. Em Java, para escrever qualquer programa é necessário definir uma nova classe com a operação `main` para representar o programa, mesmo que essa abstração de classe do programa não seja adequada para o problema específico. Uma alta demanda por abstrações dificulta o uso da notação por distrair desnecessariamente o usuário do foco principal do trabalho.

Resistência a abstrações se refere à dificuldade de definir novas abstrações na utilização da notação. Um exemplo disso são as versões iniciais de HTML (*Hyper-Text Markup Language*), onde não era possível definir novas marcações (*tags*) para representar elementos de textos com semântica definida pelo usuário. Uma alta resistência a abstrações pode tornar o

sistema limitado ou difícil de usar em casos mais complexos, onde o uso de abstrações permitem administrar melhor a complexidade do produto.

Essa dimensão apenas avalia o uso de abstrações pela notação, mas não avalia a adequação dessas abstrações para o problema ou a clareza delas, que é melhor expressada através de outras dimensões a serem vistas posteriormente, como proximidade de mapeamento, expressividade de papéis e consistência.

De forma geral, os mecanismos de abstração influenciam a flexibilidade facilitando o planejamento e descrição das modificações e permitindo estruturar o programa de forma a acomodar melhor certas modificações. Apesar de abstrações serem um recurso útil para gerenciar a complexidade do produto, em modificações exploratórias, onde erros e baixa qualidade são mais toleráveis, os mecanismos de abstração podem gerar distrações.

Os mecanismos de abstração estão mais relacionados ao projeto específico de uma linguagem e o seu uso em uma linguagem não são influenciados diretamente pelas características de linguagens dinâmicas. Contudo, o suporte a reflexão computacional tipicamente define novas abstrações para representação do meta-nível, como é o caso da reificação de elementos da linguagem. Adicionalmente, a reflexão computacional pode ser usada para implementar mecanismos de abstração adicionais. De forma geral, como a tipagem dinâmica não exige a declaração de tipo, essa abstração tipicamente não existe em linguagens dinâmicas.

2.4.8 Operações Complicadas

Essa dimensão se refere à exigência que o usuário realize atividades mentais difíceis, ou seja, que demandam muitos recursos cognitivos como lembrança e raciocínio. Exemplos de operações complicadas são operações que manipulam múltiplos níveis de indireção como operações de inserção e remoção de elementos em uma estrutura em árvore implementada usando estruturas com ponteiros em C.

Assim como a viscosidade avalia o custo mecânico da incorporação de uma mudança, operações complicadas avaliam o custo mental do planejamento e incorporação da mudança. Ou seja, mesmo que os passos para uma mudança sejam poucos ou pequenos (baixa viscosidade) eles podem ser difíceis de serem trabalhados mentalmente (muitas operações complicadas). Operações complicadas são igualmente prejudiciais para a flexibilidade, tanto em modificações definitivas como exploratórias.

O uso de reflexão computacional é tipicamente uma operação complicada, pois envolve a manipulação de valores de forma indireta através de elementos

reificados do meta-nível, como por exemplo, a manipulação de uma tabela através de sua meta-tabela que representa seu comportamento reificado. Entretanto, linguagens dinâmicas não exigem necessariamente a utilização dos mecanismos de reflexão. Em geral, o suporte a reflexão é um recurso opcional que pode ser utilizado pelo programador nos casos necessários. A tipagem estática pode auxiliar o programador a identificar erros em operações complicadas, entretanto essas características são melhores representadas em outras dimensões como dependências ocultas e propensão a erros.

A gerência de memória em um programa pode se tornar consideravelmente complexa, especialmente quando há um número grande de componentes que trocam informações armazenadas em porções de memória alocada dinamicamente. Nesse sentido, a atividade de projetar adequadamente as regras e mecanismos para que as porções de memória não utilizada sejam liberadas adequadamente e as porções de memória liberadas não sejam mais utilizadas pelas diferentes partes do programa pode ser vista como uma operação complicada a ser realizada pelo desenvolvedor. A gerência automática de memória das linguagens dinâmicas evita esse tipo de operação, pois permite que essa tarefa seja feita automaticamente pelo ambiente de execução oferecido pela linguagem.

2.4.9 Visibilidade

Essa dimensão se refere à facilidade de visualizar as diferentes partes do artefato; em particular, de poder comparar visualmente partes distintas do artefato (também chamado de *justaposição*). Um exemplo de visibilidade pobre é a separação de declaração de funções e suas respectivas implementações em arquivos diferentes, como é comumente feito em programas em C e C++. A assinatura das funções geralmente fica em um arquivo de cabeçalho (*e.g.* `.h`) e a implementação fica em um arquivo a parte (*e.g.* `.cpp`). No caso de linguagens de programação, a visibilidade pode ser amplamente influenciada pelos recursos do editor ou ambiente de desenvolvimento (IDE, do inglês *Integrated Development Environment*) utilizado. Por exemplo, muitos editores de texto permitem visualizar diferentes arquivos simultaneamente (justaposição) ou mesmo diferentes porções de um mesmo arquivo.

A visibilidade facilita o entendimento do sistema, pois permite analisar novas partes comparando-as com outras já conhecidas. Isso é particularmente importante em modificações exploratórias, onde o foco da atividade está no aprendizado. A visibilidade também pode facilitar modificações de uma forma geral, pois facilita a reutilização de porções do sistema. Por exemplo,

o programador pode visualizar uma parte do sistema sobre a qual ele quer se basear para desenvolver uma nova parte ou realizar uma modificação. Note que essa reutilização não se restringe a código, onde geralmente é preferível evitar a duplicação de código, mas pode se referir a padrões de codificação, escolha de nomes de variáveis e funções, etc.

Essa é uma dimensão pouco relacionada às características de linguagens dinâmicas, pois é mais influenciada pela sintaxe da linguagem e o suporte oferecido por sub-dispositivos utilizados, como editores de texto e IDEs. Por outro lado, programas escritos em linguagens dinâmicas tendem a apresentar uma melhor visibilidade por serem menores do que programas escritos em linguagens que se baseiam em declarações explícitas de tipo. Entretanto, essa é uma característica melhor representada pela dimensão seguinte.

2.4.10 Prolixidade

A prolixidade se caracteriza pela redundância ou excesso de informações ou representações que podem ofuscar outros itens mais relevantes. Um exemplo disso é o uso de palavras reservadas longas ou que precisam ser repetidas muitas vezes. Por exemplo, em C++ o escopo de acesso dos membros de uma classe é feito em seções iniciadas por uma palavra reservada (figura 2.8(a)), enquanto que em Java, cada membro deve ter uma palavra indicando seu escopo, que é repetida muitas vezes (figura 2.8(b)). Declarações explícitas de tipo também podem ser vistas como um exemplo de prolixidade em linguagens de programação, pois em algumas situações elas acrescentam pouca informação ao desenvolvedor, sendo principalmente um recurso que facilita a tarefa do compilador.

Figura 2.8: Comparação de prolixidade na declaração de classes em C++ e Java.

2.8(a): Classe em C++.

```
1 class MyClass { // private:
2   string field1;
3   string field2;
4   string field3;
5 public:
6   void meth1() {/* ... */};
7   void meth2() {/* ... */};
8   void meth3() {/* ... */};
9 };
```

2.8(b): Classe em Java.

```
1 class MyClass {
2   private String field1;
3   private String field2;
4   private String field3;
5
6   public void meth1() {/* ... */};
7   public void meth2() {/* ... */};
8   public void meth3() {/* ... */};
9 };
```

A prolixidade dificulta a flexibilidade indiretamente, pois tende a reduzir a visibilidade do sistema devido ao excesso de informações e construções que podem impedir a visualização de elementos de maior interesse do programador. Além disso, linguagens prolixas também tendem a ser mais viscosas se as

informações em excesso ou redundantes são mantidas pelo programador sem auxílio de sub-dispositivos que facilitem essa tarefa, como é o exemplo de IDEs para Java, que apresentam funcionalidades importantes[2] comumente utilizadas por programadores na modificação de suas aplicações (Murphy et al., 2006).

Apesar da ausência de declarações explícitas de tipo geralmente contribuir para reduzir a prolixidade, essa não é uma peculiaridade de linguagens dinâmicas, visto que mecanismos como inferência de tipo podem diminuir consideravelmente a prolixidade causada por declarações explícitas em linguagens estaticamente tipadas. De forma geral, as características das linguagens dinâmicas não influenciam diretamente na prolixidade, que está mais diretamente ligada à gramática e às construções específicas da linguagem.

2.4.11 Expressividade de Papéis

Essa dimensão se refere à clareza do propósito de cada entidade ou parte do artefato, ou seja, se é possível identificar facilmente a semântica de cada elemento da notação. Por exemplo, em Lua, tabelas são usadas indistintamente em diversas situações, seja para implementar estruturas de dados, objetos com comportamento, meta-mecanismos (*i.e.* meta-tabelas), elementos reificados (*e.g.* o ambiente global de uma função), etc. Em muitas situações não é claro o propósito ou a semântica de uma estrutura criada como uma tabela sem analisar todos os seus usos.

A expressividade de papéis tem influência direta na facilidade de entendimento e aprendizado do sistema, sendo uma dimensão importante em modificações exploratórias. De forma geral, uma boa expressividade é importante para facilitar a identificação do papel de cada elemento sendo examinado, alterado ou incluído durante as modificações. Em modificações definitivas, essa dimensão tem menos importância, pois como as modificações são menos intensas e abrangentes, é mais fácil utilizar sub-dispositivos que permitam melhorar a expressividade, como a documentação da implementação.

Declarações de tipo podem ser utilizadas para expressar o papel de cada valor. Por isso, a tipagem dinâmica pode ser considerada um fator de redução da expressividade em linguagens. Entretanto, os papéis dos elementos podem ser melhor representados por outras abstrações além de tipos de dados. Por exemplo, Java faz uma distinção entre classe (tipo e implementação) e interface (apenas o tipo), enquanto que em C++ há apenas a primeira forma para representar as duas coisas. No caso de Java, há duas construções distintas para

[2]Uma enquete feita pela Internet, disponível em `http://today.java.net/pub/pq/111`, aponta que 73.9% dos 886 participantes afirmam que teriam dificuldades ou seriam incapazes de programar em Java sem suporte de uma IDE.

representar os dois papéis, enquanto que em C++, a mesma construção (classe) é usada para duas coisas diferentes, não deixando claro, à primeira vista, qual dos dois papéis está representando. Por essas razões, podemos considerar que a expressividade de papéis está pouco relacionada às características de linguagens dinâmicas, sendo mais diretamente influenciada pelas abstrações disponíveis na linguagem ou nos mecanismos de abstração disponíveis.

2.4.12 Proximidade de Mapeamento

Essa dimensão se refere à adequação da representação do que está sendo modelado ao seu domínio específico. Por exemplo, uma linguagem lógica como Prolog apresenta maior proximidade com máquinas de inferência baseadas em proposições lógicas do que uma linguagem como PostScript, que apresenta maior proximidade com a geração de documentos. Basicamente, linguagens de domínio específico são mais próximas ao seu domínio de aplicação do que linguagens de propósito geral.

Quanto mais próximo é o mapeamento da linguagem ao sistema sendo desenvolvido, mais fácil e naturalmente o sistema será descrito através dela. Isso ajuda tanto na introdução das modificações como no seu planejamento. Mais especificamente, o aprendizado do sistema é mais fácil quando há uma boa proximidade da linguagem com o domínio de aplicação. Inclusive, o próprio sistema pode auxiliar no ensino do domínio de aplicação (Colgan, 2000). Por essa razão, a proximidade de mapeamento é particularmente interessante em modificações exploratórias.

De forma geral, as características das linguagens dinâmicas não influenciam diretamente a proximidade do mapeamento, que está mais relacionada às abstrações e construções específicas da linguagem de programação. Contudo, a facilidade de introduzir novas abstrações através de reflexão computacional pode facilitar a aproximação da linguagem ou do sistema sendo desenvolvido (artefato) a um dado domínio. Além disso, em muitos domínios a memória para armazenamento de informações temporárias ou mais perenes não é considerada explicitamente, por isso a gerência automática de memória também pode ser vista como uma maior aproximação com o domínio nesse caso. Em particular, linguagens dinâmicas apresentam boa proximidade com domínios dinâmicos, como manipulação de dados não estruturados e integração dinâmica de ferramentas e sistemas (Ousterhout, 1998).

2.4.13 Consistência

A consistência se refere à similaridade sintática de formas com semânticas similares. Um exemplo disso, é a atribuição em Visual Basic, que é feita através da forma `var = valor`, entretanto, se o valor sendo atribuído for um objeto, é necessário utilizar a forma `Set var = valor`, caso contrário apenas uma propriedade (denominada propriedade *default*) é copiada (Schentag). Um exemplo inverso é o uso da palavra `static` em C++, que é utilizada com diferentes significados dependendo do contexto (Hammer).

A consistência é importante para a flexibilidade, não apenas por facilitar o entendimento da linguagem ou do sistema, facilitando assim sua modificação, mas também por facilitar a reutilização de trechos e idéias através da implementação. Se a linguagem ou o sistema não é consistente, é necessário adaptar idéias de uma parte para aplicar em outras, que pode se mostrar mais propenso a erros. Portanto, pouca consistência é ruim em modificações exploratórias, por dificultar o entendimento do sistema, assim como é ruim em modificações definitivas, por propiciar maiores alterações no código e maior possibilidade de erros.

A sintaxe específica da linguagem é muito mais influente na sua consistência do que as características de linguagens dinâmicas. Por outro lado, através da reflexão computacional o programador pode modificar a semântica de alguns aspectos da linguagem, permitindo assim comprometer sua consistência.

2.4.14 Notação Secundária

Essa dimensão se refere ao suporte à uma notação auxiliar que permita que o usuário possa descrever algo que não é considerado pela notação. Idealmente, a notação secundária tem semântica definida pelo usuário e não modifica a interpretação da notação principal. Exemplos de notação secundária incluem a possibilidade de comentários e anotações em um código, separação de grupos de comandos ou declarações usando linhas em branco e padrões de indentação em linguagens textuais[3].

Em geral, o suporte a notação secundária pode melhorar a flexibilidade por permitir que o usuário crie sub-dispositivos que permitam facilitar a compreensão do sistema, como é o caso de documentação no código na forma de

[3]Em linguagens como Python, em que a indentação tem semântica definida pela linguagem e não pode ser utilizada arbitrariamente pelo usuário, padrões de indentação não se caracterizam como notação secundária.

comentários. Contudo, como a notação secundária é geralmente livre e mantida pelo usuário sem o auxílio de sub-dispositivos, a informação descrita pode ficar desatualizada, incompleta ou mesmo errada. Portanto, apesar de ser um mecanismo paliativo para possíveis deficiências da notação principal, a notação secundária pode introduzir problemas que dificultem a modificação do sistema, ou seja, sua flexibilidade. Por outro lado, o suporte a notação secundária pode ser um mecanismo valioso no estudo do sistema.

O suporte a notações secundárias está diretamente relacionado ao projeto específico de uma linguagem de programação e tipicamente não é influenciado pelas características de linguagens dinâmicas.

Dependendo do tipo de atividade sendo avaliada com o CDN, as diferentes dimensões podem ter relevâncias diferentes. A figura 2.9 indica a relevância das dimensões utilizadas neste trabalho para cada tipo de modificação considerada neste trabalho. Cada dimensão é vista como positiva ou negativa para a flexibilidade do software tanto quando são consideradas modificações definitivas como exploratórias. Entretanto, para cada tipo de modificação a intensidade dessa relevância pode ser mais ou menos acentuada.

Figura 2.9: Relevância das dimensões cognitivas em cada tipo de modificação.

Dimensão Cognitiva	Mod. Definitiva	Mod. Exploratória
Viscosidade	muito ruim	muito ruim
Dependências Ocultas	ruim	ruim
Compromissos Prematuros	ruim	ruim
Mecanismos de Provisão	bom	muito bom
Avaliação Progressiva	bom	muito bom
Propensão a Erros	muito ruim	ruim
Operações Complicadas	muito ruim	muito ruim
Barreira de Abstração	ruim	ruim
Demanda de Abstrações	tolerável	ruim
Resistência a Abstrações	ruim	tolerável
Visibilidade	bom	muito bom
Prolixidade	ruim	ruim
Expressividade de Papéis	bom	muito bom
Proximidade de Mapeamento	bom	muito bom
Consistência	bom	muito bom
Notação Secundária	bom	bom

A figura 2.10 apresenta um resumo de como as características das linguagens dinâmicas influenciam a flexibilidade do software em relação às dimensões do CDN apresentadas neste capítulo. Através dessa figura podemos ver que o impacto das características das linguagens dinâmicas é positivo na

maioria das dimensões do CDN, inclusive se levarmos em consideração as dimensões mais relevantes para cada tipo de modificação. Portanto, podemos ver as linguagens dinâmicas como potencialmente mais flexíveis do que linguagens pouco dinâmicas. Entretanto, o impacto das características das linguagens dinâmicas varia de acordo com cada aplicação. Por exemplo, em um programa cujo sistema de tipos é simples, a menor prolixidade devido a falta de declarações explícitas de tipo devido a tipagem dinâmica se mostra pouco relevante. No capítulo 4 é feita uma análise de como as características das linguagens dinâmicas contribuem na construção do middleware OiL, em comparação com outra implementação de middleware em uma linguagem menos dinâmica.

Figura 2.10: Aspectos positivos e negativos de ling. dinâmicas na flexibilidade.

	Aspectos Positivos	Aspectos Negativos
Interpreta.	Menor Viscosidade[1,2] Melhor Avaliação Progressiva[2]	
Tipagem Dinâmica	Menor Viscosidade[1,2] Menos Comprom. Prematuros Mais Mecanismos de Provisão[2] Melhor Avaliação Progressiva[2] Menor Barreira de Abstração Menor Demanda de Abstrações Maior Visibilidade[2] Menor Prolixidade	Mais Dependências Ocultas Maior Propensão a Erros[1] Menor Expressividade de Papéis
Reflexão Computac.	Menor Viscosidade[1,2] Menos Comprom. Prematuros Mais Mecanismos de Provisão[2] Menor Resistência a Abstrações Maior Expressividade de Papéis[2] Maior Prox. de Mapeamento[2]	Mais Operações Complicadas[1,2] Maior Barreira de Abstração Maior Demanda de Abstrações Menor Consistência[2]
Gerência Automát. de Memo.	Menor Viscosidade[1,2] Menor Propensão a Erros[1] Menos Operações Complicadas[1,2] Menor Barreira de Abstração Maior Prox. de Mapeamento[2]	

[1]Mais relevante em modificações definitivas
[2]Mais relevante em modificações exploratórias