Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Disciplina: DIM0437 — Linguagens de Programação: Conceitos e Paradigmas
Docente: Umberto Souza da Costa
Discentes: Dogival Ferreira da Silva Junior
Felipe Cortez de Sá
Gabriel Sebastian von Conta
Phellipe Albert Volkmer
Vinícius Araújo Petch

Subproblema 8 — Abstrações

1 Problema

1.1 Produto do problema

Definição de abstrações de larga escala presentes na linguagem de programação a ser definida pelo grupo, considerando o contexto de aplicação dos programas nela escritos e características principais da mesma.

1.2 Questões

1. Qual o conceito de abstração?
2. Quais serão as formas de abstrações presentes na linguagem proposta:
   (a) Tipos abstratos de dados?
   (b) Classes de objetos?
   (c) Pacotes?
   (d) Módulos?
3. O que caracteriza a categorização de abstrações mostrada acima? Quais são as semelhanças e diferenças dos conceitos definidos na questão anterior?
4. Como os conceitos discutidos poderiam ser implementados em sua linguagem?
5. Além dessas características, como sua linguagem poderia incluir a noção de genericidade (para tipos, classes, pacotes ou módulos)?
6. Como sua linguagem lidará com construções de encapsulamento e com o encapsulamento de nomes?

2 Resoluções

1. Uma **abstração** é uma simplificação útil para o programador, uma vez que permite representar uma entidade, filtrando as informações e operações, aproveitando apenas o que é relevante.

   Por exemplo, uma entidade `grupo de lpcp` possui os atributos gerais componentes e nota e trabalhos, mas também possui características únicas a certos grupos, como por exemplo `possui gordos`. Criamos então um objeto `G2` que precisa do atributo específico `possui gordos`, mas não `componentes`, `nota` e `trabalhos`, herdados da entidade `grupo de lpcp`.

2. (a) No caso de tipos abstratos de dados:

   Um *tipo abstrato de dado* tem a representação do tipo dos objetos ocultas para as unidades de programa que utilizam esse tipo. Dessa forma as únicas operações possíveis nesses objetos são aquelas que estão na definição do objeto.

   Além disso as operações do tipo e as operações definidas nos objetos desse tipo, que fornecem a interface desse tipo, são contidas em uma única unidade sintática.

   Assim, a interface do tipo não depende da representação dos objetos nem da definição de suas operações. Além disso outras unidades de programa podem usar variáveis do tipo definido.

   Na nossa linguagem, utilizaremos tipos abstratos de dados, presentes tanto no tipo float, quanto em tipos criados pelo usuário.

   (b) No caso de pacotes:

   Um *pacote* é como a implementação de módulo é chamada na linguagem Ada. Nela, os cabeçalhos e corpos dos módulos podem ser separados.

   Nós utilizaremos pacotes (packages). A ideia de pacotes será implementada semelhante à utilização de módulos em Ada, onde cabeçalho e corpo podem ser separados. Porém, na nossa implementação cabeçalho e corpo terão que estar no mesmo arquivo.

   (c) No caso de classes:

   Uma *classe* é um *template* extensível para a criação de objetos. As classes podem ser instanciadas através de uma chamada para o método construtor, uma subrotina responsável pela inicialização de dados de objetos. As entidades declaradas como públicas podem ser acessadas através dessas instâncias. De acordo com Sebesta, esta maneira é mais limpa e direta que o encapsulamento em Ada.

   Classes possuem membros de dados, que armazenam estado e informação, e membros de subrotina, implementações de comportamento, também chamados de métodos em linguagens orientadas a objeto. Além disso, membros de dados podem ser categorizados em membros de classe, independentes de instanciação e membros de instância, que podem assumir valores diferentes por cada objeto. Para organizar o código e aumentar a legibilidade, C++ permite separar a definição de uma classe e sua implementação.

A diferença entre as classes das linguagens orientadas a objetos e pacotes em Ada é que classes geram tipos.

(d) No caso dos módulos:

Um *módulo* é uma abstração que tem como como finalidade a granularização do código, é implementado de forma a dividir as interfaces e as implementações em arquivos diferente assim como na ocorre na linguagem C. Ao final permite uma melhor organização do código e compilação separada.

Na nossa linguagem não utilizaremos módulos, pois como ele se destina a um público iniciante, o caráter dos programas é simples, não necessitando das abstrações de módulo.

3. Todas são encapsulamentos que permitem a abstração de dados e funções do código. No caso do tipo de dados, temos a noção de uma implementação de um grupo de funções, das quais o usuário só conhece aquelas que estão na definição do grupo. Classes de objetos são uma implementação em cima dessa noção criada a partir de TAD.

   No caso de pacotes e módulos, ambos tem a função tanto de organização do código e separação da compilação. A diferença se encontra na implementação de ambos.

4. Em relação aos pacotes, como foi dito anteriormente, eles serão implementados no mesmo estilo da Ada, porém com cabeçalho e corpo no mesmo arquivo. Para chamar um pacote em um arquivo, será utilizado o comando **`import <nome_do_arquivo>`**

   Anotações - Criação de pacotes - porém, códigos monolíticos não são muito problema

5. Anotações - Tipos e pacotes mais genéricos, tipos parametrizados

   Parametrização de um TAD se refere a tornar tal tipo mais genérico. Por exemplo, ser capaz de definir uma lista de um TAD que seja capaz de guardar qualquer tipo de variável escalar e ser de qualquer tamanho, ao invés de possuir uma variável escalar fixa e ter um tamanho fixo. O estilo escolhido foi a implementação de parametrização em C++. Nele, é possível parametrizar uma TAD (ou classe) apenas alterando o construtor, ou adicionando um construtor genérico (que pode ser feito caso a linguagem possua sobrecarga, que é o nosso caso). Porém, como não possuímos classes, então tal parametrização de aplica apenas a TADs.

6. Quando definimos tipos abstratos de dados, classes, módulos e pacotes, usamos de encapsulamentos mínimos, ou seja, coleções pequenas de código e data logicamente relacionado, de modo a criar uma abstração para o usuário. Essas coleções são chamadas de encapsulamento. Mas quando o código começa a crescer na casa das milhares de linhas de código, utilizamos esse conceito de forma expandida.

   Em C, uma coleção de funções e dados relacionados podem ser colocadas em arquivos compilados separadamente, agindo como uma biblioteca. A interface para tal arquivo é colocada em um arquivo separado, chamado de header file (arquivo de cabeçalho). Quando o pré-processador lê uma declaração de **`#include`**, como **`#include "biblioteca.h"`**, o conteúdo do arquivo de cabeçalho (neste caso, **`biblioteca.h`**) é inserido. Esse tipo de encapsulamento gera algumas inseguranças. Por exemplo, o código do *header* pode ser simplesmente copiado e colado, em vez de usar o **`#include`**. Inicialmente, isso não faz diferença, porém quando o autor da biblioteca faz alterações, o *linker* pode não

detectar a incompatibilidade de tipos entre as definições antigas (copiadas e coladas) e as implementações novas.

Decidimos adotar o mesmo funcionamento para nossa linguagem, apesar da insegurança, já que o conceito é simples e útil, e não temos como objetivo dar suporte ao desenvolvimento de aplicações muito grandes. Pelo mesmo motivo, não teremos encapsulamento de nomes, pois os programas não devem ser grandes o suficiente para poder causar conflito de nomes.