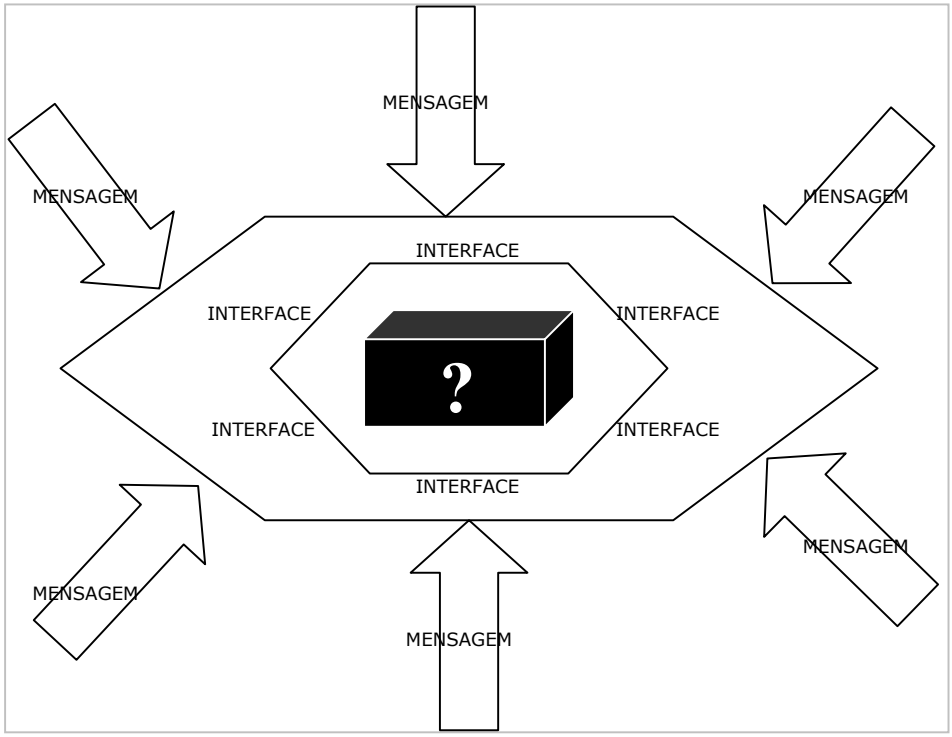
Capítulo 2 - Encapsulamento

1. Encapsulamento: "Não mostre as cartas do seu baralho"

1.1 O que é o encapsulamento?

Em vez de ver um programa como uma única entidade grande e monolítica, o encapsulamento permite que você o divida em várias partes menores e independentes. Cada parte possui implementação e realiza seu trabalho independentemente das outras partes. O encapsulamento mantém essa independência, ocultando os detalhes internos, ou seja, a implementação de cada parte, através de uma interface externa.

O encapsulamento permite a visualização de uma entidade de software como uma caixa preta. Você sabe o que a caixa preta faz, pois conhece sua interface externa e, simplesmente, envia mensagens para a caixa preta, não se preocupando com o que acontece dentro dela; você apenas se preocupa com o fato de que isso aconteça.



Uma *interface* lista os serviços fornecidos por um componente. A interface é um contrato com o mundo exterior, que define exatamente o que uma entidade externa pode fazer com o objeto. Uma interface é o painel de controle do objeto. Talvez você esteja familiarizado com o termo de programação API (Interface de Programa Aplicativo). Uma *interface* é semelhante a API para um objeto. A interface lista todos os métodos e argumentos que o objeto entende.

A *implementação* define como um componente realmente fornece um serviço e os detalhes internos desse componente.

Veja, a seguir um exemplo de interface e implementação.

Actionscript

```
/*
 * Relata mensagens de depuração, informativas, de alerta e de erro
 * */
class Log {
	public function debug(message:String) {
		imprime("DEBUG", message);
	}
	public function info(message:String) {
		imprime("INFO", message);
	}
	public function warning(message:String) {
		imprime("WARNING", message);
	}
	public function error(message:String) {
		imprime("ERROR", message);
	}
	public function fatal(message:String) {
		imprime("FATAL", message);
	}
	private function imprime(message:String, severity:String) {
		trace(severity+": "+message);
	}
}
```

Java

```
/*
 * Relata mensagens de depuração, informativas, de alerta e de erro
 * */
public class Log {
	public void debug( String message )
	{
		imprime( "DEBUG", message);
	}
	public void info( String message )
	{
		imprime( "INFO", message);
	}
	public void warning( String message )
	{
		imprime( "WARNING", message);
	}
	public void error( String message )
	{
		imprime( "ERROR", message);
	}
	public void fatal( String message )
	{
		imprime( "FATAL", message);
	}
	private void imprime( String message, String severity)
	{
		System.out.println(severity + ": "+ message);
	}
}
```

A classe *Log* contém objetos para relatar mensagens de depuração, informativas, de alerta e de erro, durante a execução. A interface da classe *Log* é constituída de todos

os comportamentos disponíveis para o mundo exterior. Os comportamentos disponíveis para o mundo exterior são conhecidos como interface pública. A interface pública da classe *Log* inclui os seguintes métodos:

- public void debug(String message)
- public void info(String message)
- public void warning(String message)
- public void error(String message)
- public void fatal(String message)

Tudo mais na definição da classe, além dessas cinco declarações de método, é implementação. Lembre-se de que a implementação define como algo é feito, ao contrário a interface define o contrato com o mundo exterior.

1.2 Níveis de acesso

Há dois principais níveis de acesso no Flash, que são:

1. Público (public)
   - Garante acesso a todos os objetos.
2. Privado (private)
   - Garante o acesso apenas para a instância, ou seja, para aquele objeto.

1.3 Por que encapsular?

Quando usado cuidadosamente, o encapsulamento transforma seus objetos em componentes "plugáveis". Para que outro objeto use seu componente, ele só precisa saber como usar a interface pública do componente. Tal independência tem três vantagens importantes:

1. Independência significa que você pode reutilizar o objeto em qualquer lugar. Quando você encapsular corretamente seus objetos, eles não estarão vinculados a nenhum programa em particular. Em vez disso, você pode usá-los sempre que seu uso fizer sentido. Para usar o objeto em qualquer lugar, você simplesmente exerce sua interface.
2. O encapsulamento permite que você torne transparentes as alterações em seu objeto. Desde que você não altere sua interface, todas as alterações permanecerão transparentes para aqueles que estiverem usando o objeto. O encapsulamento permite que você atualize seu componente, forneça uma implementação mais eficiente ou corrija erros – tudo isso sem ter de tocar nos outros objetos de seu programa. Os usuários de seu objeto se beneficiarão automaticamente de todas as alterações que você fizer.
3. Usar um objeto encapsulado não causará efeitos colaterais inesperados entre os objetos e o restante do programa. Como o objeto tem implementação independente, ele não terá nenhuma outra interação com o restante do programa, além de sua interface.

1.4 Três características do encapsulamento eficaz

As três características do encapsulamento eficaz são:

- Abstração
- Ocultação da implementação
- Divisão da responsabilidade

Abstração

Abstração é o processo de simplificar um problema difícil. Quando se começa a resolver um problema, você não se preocupa com cada detalhe. Em vez disso, você o simplifica, tratando apenas dos detalhes pertinentes a uma solução.

A abstração possui duas vantagens. Primeiro, ela permite que você resolva um problema facilmente. Mais importante, a abstração o ajuda a obter reutilização, característica que deve ser indispensável quando se pensa em Flash, já que precisamos de arquivos leves e eficientes. Muitas vezes, os componentes de software são demasiadamente especializados. Essa especialização, combinada com uma interdependência desnecessária entre os componentes, torna difícil reutilizar um código existente em outra parte. Quando possível você deve se esforçar por criar objetos que possam resolver um domínio inteiro de problemas. A abstração permite que você resolva um problema uma vez e depois use essa solução por todo o domínio desse problema.

Embora seja desejável escrever código abstrato e evitar uma especialização demasiada, é complexo escrever código em um nível que combine estas duas características, especialmente quando você está começando a praticar POO. Existe uma linha tênue entre muita e pouca especialização. Essa linha pode ser discernida apenas com a experiência. Entretanto, você precisa saber desse poderoso conceito.

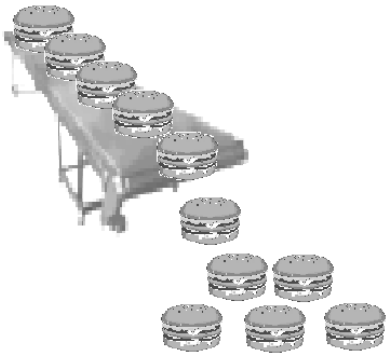
Vejamos a seguir dois exemplos de abstração:

1. Exemplo de uma fila de banco

   Imagine pessoas em uma fila de banco, esperando por um caixa. Assim que um caixa se torna disponível, a primeira pessoa da fila avança para a janela aberta. As pessoas sempre deixam a fila na ordem em que entraram nela, ou seja, o primeiro a entrar é o primeiro a sair (*first in, first out* – FIFO – conceito de fila), sendo que essa ordem é sempre mantida.

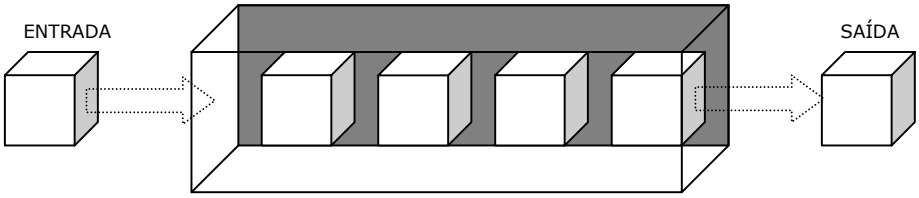
2. Exemplo de montagem de sanduíches em um *fast food*

Considere um estabelecimento de sanduíches do tipo *fast food*. Quando um novo sanduíche fica pronto, ele é colocado atrás do último sanduíche que está no escaninho; desse modo, o primeiro sanduíche retirado também é o mais antigo (o primeiro que entra é o primeiro que sai), FIFO é o esquema do restaurante.



Embora cada um desses exemplos seja específicos, você pode encontrar uma descrição genérica que funcione em cada situação. Ou seja, você pode chegar a uma abstração.

Cada domínio é um exemplo de fila do tipo primeiro a entrar, primeiro a sair (FIFO). Não importa quantos elementos apareçam na fila. Importa que os elementos entram no final da fila e saem dela a partir da frente, conforme ilustrado abaixo:



Abstraindo os domínios, você pode criar uma fila uma vez e reutilizá-la em qualquer problema que modele um domínio onde exista uma ordenação FIFO de elementos.

Para obter uma abstração eficaz você deve seguir as seguintes dicas:

1. Trate do caso geral e não do caso específico. A abstração pode não saltar à sua frente na 1ª, 2ª ou 3ª vez que você resolver um problema que está sujeito a ser abstraído. Portanto não caia na paralisia da abstração. O uso incorreto da abstração pode levá-lo a: prazos finais perdidos e abstração incorreta.
2. Ao confrontar vários problemas procure o que for comum a todos. (enxergar um conceito). Mas para não se perder tenha em mente o objetivo de resolver seu problema primeiro.
3. Não se esqueça de que você tem um problema a resolver. A abstração é valiosa, mas não descuide do problema na esperança de escrever código abstrato. Veja a abstração como bônus e não como um objetivo final.

4. Prepare-se para a falha. É quase impossível escrever uma abstração que funcione em todas as situações. Para a abstração a experiência é um ponto muito importante.

Ocultação da implementação

A abstração é apenas uma característica do encapsulamento eficaz. Você pode escrever código abstrato que não é encapsulado. Em vez disso, você também precisa ocultar as implementações internas de seus objetos.

A ocultação da implementação tem duas vantagens:

1. Ela protege o seu objeto de seus usuários;
2. Ela protege os usuários de seu objeto do próprio objeto;

Tipo Abstrato de Dados – TAD

Explorando a primeira vantagem podemos proteger seu objeto através da utilização do Tipo Abstrato de Dados (TAD). O TAD foi introduzido através da linguagem Simula em 1966. Na verdade, os TADs são decididamente não Orientados a Objetos; em vez disso, eles formam um sub conjunto da Orientação a Objetos. Entretanto, os TADs apresentam duas características interessantes: *abstração* e *tipo*. É essa idéia de tipo que é importante, pois sem ela, você não pode ter um verdadeiro encapsulamento (que é imposto em nível de linguagem, através de construções internas da linguagem).

Um TAD é um conjunto de dados e um conjunto de operações sobre esses dados. Os TADs permitem que você defina novos tipos na linguagem, ocultando dados internos e o estado, através de uma interface bem definida. Essa interface apresenta o TAD como uma única unidade atômica (ou seja, uma unidade independente – pensando na soma de dois inteiros, você pensa apenas a respeito da adição de dois números; mesmo que os bits representem o inteiro, a linguagem de programação apresenta o inteiro apenas como um número para o programador).

O que é um tipo?

Os tipos definem as diferentes espécies de valores que estão disponíveis para seus programas. Exemplos de alguns tipos comuns são: *Number, Boolean, String*. Essas definições de tipo informam exatamente quais espécies de tipos estão disponíveis, o que os tipos fazem e o que você pode fazer com eles.

Usaremos a seguinte definição para tipo:

> Os *tipos* definem as diferentes espécies de valores que você pode usar em seus programas. Um tipo define o domínio a partir do qual seus valores válidos podem ser extraídos. Para inteiros positivos, são os números sem partes fracionárias e que são maiores ou iguais a zero. Para tipos estruturados, a definição é mais complexa. Além do domínio, a definição de tipo inclui quais operações são válidas no tipo e quais são seus resultados.

Pegue o exemplo do objeto *Item* do capítulo anterior. A criação da classe Item adiciona um novo tipo em seu vocabulário de programação. Em vez de pensar a respeito de uma id, uma descrição e um preço do produto como entidades separadas, provavelmente regiões desconectadas da memória ou variáveis, você pensa

simplesmente em termos de Item. Assim, os tipos permitem representar estruturas complexas em um nível mais simples e mais conceitual. Eles o protegem dos detalhes desnecessários. Isso o libera para trabalhar no nível do problema, em vez de trabalhar no nível da implementação.

Outra vantagem importante é que a definição de um tipo protege o tipo do programador. Uma definição de tipo garante que qualquer objeto que interaja com o tipo, o faça de maneira correta, consistente e segura. As restrições impostas por um tipo impedem os objetos de terem uma interação inconsistente, possivelmente destrutiva. A declaração de um tipo impede que o tipo seja usado de maneira não projetada e arbitrária. Uma declaração de tipo garante uso correto.

Pense novamente no objeto Item do primeiro capítulo. Imagine que tivéssemos alterado um pouco a definição de Item:

Actionscript

```
class UnencapsulatedItem
{
	//ATRIBUTOS - INICIO
	public var unit_price :Number = 0;
	//uma porcentagem de discount
	//que se aplica ao preço
	public var discount:Number = 0;
	public var quantity:Number = 0;
	public var description:String;
	public var id :String;
	//ATRIBUTOS - FIM

	//CONSTRUTORES - INICIO
	public function Item (id:String , description:String ,
					quantity:Number , price: Number)
	{
		this.id = id;
		this.description = description;
		this.unit_price  = price;
		setQuantity(quantity);
	}
	//CONSTRUTORES - FIM

	public  function setDiscont(valor:Number)
	{
		if ( (valor <= 1) && (valor >= 0) )
		{
			this.discount = discount;
		}else{
			this.discount = 0;
		}
	}

	public function getDiscont():Number
	{
		return discount;
	}

	public function getAdjustedTotal():Number
	{

		var total:Number = unit_price  * quantity;
		var total_discount :Number = total * discount;
```

```
            var adjusted_total :Number = total - total_discount ;

            return adjusted_total;
        }

//..
}
```

Java

```
public class UnencapsulatedItem
{
        public double unit_price;
        public double discount; //uma % de desconto a ser aplicada no preço
        public int quantity;
        public String description;
        public String id;

        public UnencapsulatedItem (String id, String description,
                                   int quantity, double price)
        {
                this.id = id;
                this.description = description;
                this.unit_price = price;
                this.quantity = quantity;
        }
        public double getAdjustedTotal()
        {
                double total = unit_price * quantity;
                double total_discount = total * discount;
                double adjusted_total = total - total_discount;

                return adjusted_total;
        }

        public void setDiscont (double value)
        {
                if ( (value <= 1) && (value >= 0) )
                {
                        this.discount = discount;
                }
                else
                {
                        this.discount = 0;
                }
        }

        public double getDiscont()
        {
                return discount;
        }

        //...
}
```

Você notará que todas as variáveis internas, agora, estão publicamente disponíveis. E se alguém escrevesse o programa a seguir, usando o novo UnencapsulatedItem:

Crie um novo documento Flash chamado “encapsulamentoIncorreto”

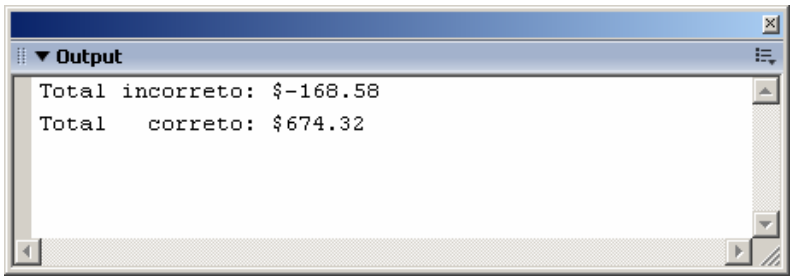
Actionscript

```
var monitor = new UnencapsulatedItem("eletronico-015", "17\" SVGA Monitor", 1,
674.32);
monitor.discount = 1.25;
//inválido, o desconto deve ser menor do que 100%
var price = monitor.getAdjustedTotal();
trace("Total incorreto: $"+price);
monitor.setDiscont(1.25);
// inválido, mas o erro é capturado
price = monitor.getAdjustedTotal();
trace("Total   correto: $"+price);
```

Java

```
public class Executor {
	/*
	 * Ponto de entrada da Aplicação
	 * */
	public static void main(String[] args) {
		UnencapsulatedItem monitor = new UnencapsulatedItem( "eletronico-
015","17\" SVGA Monitor",1,674.32);
		monitor.discount = 1.25;//inválido, o desconto deve ser menor do
que 100%
		double price = monitor.getAdjustedTotal();
		System.out.println("Total incorreto: $" + price);
		monitor.setDiscont(1.25);// inválido, mas o erro é capturado
		price = monitor.getAdjustedTotal();
		System.out.println("Total   correto: $" + price);
	}
}
```

A figura abaixo mostra o que acontece quando é executado o arquivo de teste "encapsulamentoIncorreto".



Abrindo o tipo UnencapsulatedItem para acesso liberado, outros podem chegar e deixar uma instância desse objeto em um estado inválido. Nesse caso, *o documento FLA* cria uma instância desse objeto e aplica diretamente um desconto inválido. O resultado é um preço negativo ajustado.

Podemos usar os métodos “set” e “get” para fazer o encapsulamento, exemplo:

Sem “get” e “set”

```
function getNome():String {
        return nomeUsuario;
}
function setNomeUsuario(nome:String):Void {
        nomeUsuario = nome;
}
// Chamando método
var nome = obj.getNome();
// Chamando método
obj.setNomeUsuario("João");
```

Com os métodos “get” e “set”

```
function get usuario():String {
  return nomeUsuario;
}
function set usuario(nome:String):Void {
  nomeUsuario = nome;
}

//Modo de acesso
var nome = obj.usuario;
obj.usuario = "João";
```

Obs:

- O método get não pode ter parâmetros
- O nome do método get pode ter o mesmo nome do método set no mesmo escopo.

Divisão da responsabilidade

Divisão da responsabilidade correta significa que cada objeto deve executar uma função – sua responsabilidade – e executá-la bem.

A divisão da responsabilidade correta leva ao objeto coesivo. Ou seja, não faz sentido encapsular muitas funções aleatórias e variáveis. Há a necessidade de um forte

vínculo conceitual entre si. Todas as funções devem trabalhar no sentido de uma responsabilidade comum.

Ocultação da implementação e divisão da responsabilidade andam lado a lado.

Se a implementação ficar aberta para o mundo exterior, um usuário poderá começar a atuar diretamente na implementação – duplicando assim a responsabilidade.

Assim que dois objetos começam a fazer a mesma tarefa, você sabe que não tem uma divisão correta de responsabilidades. Quando aparecer a lógica redundante é preciso refazer o código. Refazer o código é parte esperada do ciclo de desenvolvimento OO.

O encapsulamento é como o gerente eficiente. Como no mundo real, conhecimento e responsabilidade precisam ser delegados para aqueles que sabem como fazer o trabalho da melhor forma possível.

Um exemplo de erro de atribuição de responsabilidade seria um objeto *Item* que não soubesse calcular o total com desconto. Sendo assim, essa responsabilidade seria passada para o objeto que utilizasse Item, de maneira que esse objeto fosse o responsável em dizer para o objeto Item como calcular o total sempre que necessário, tendo assim uma ação como um gerente ineficiente.

Chamar ou executar várias funções para calcular o total ajustado, retira a responsabilidade do item e a coloca nas mãos do usuário. Retirar a responsabilidade dessa maneira é tão ruim quanto expor implementações internas. Você permite que a responsabilidade duplicada se espalhe por todo o seu código e para cada objeto que desejar calcular o total ajustado, precisará repetir a lógica do cálculo.

Quando você tem objetos que não dividem corretamente a responsabilidade, cria-se o código procedural, centrado nos dados. Se você simplesmente enviar uma mensagem para um objeto e confiar que ele faça seu trabalho, esse é o verdadeiro desenvolvimento orientado a objetos.

O encapsulamento está diretamente ligado à ocultação de detalhes. A responsabilidade coloca o conhecimento de certos detalhes no lugar ao qual eles pertencem. É importante que os objetos tenham apenas uma ou um pequeno número de responsabilidades. Se um objeto possui um excesso de responsabilidades, sua implementação se tornará muito confusa e difícil de manter e entender. Para alterar uma responsabilidade, você correrá o risco de alterar outro comportamento inadvertidamente. Ele também centralizará muito conhecimento, que seria melhor gerenciado se fosse espalhado. Quando um objeto fica grande demais, ele quase se torna um programa completo e cai nas armadilhas procedurais. Como resultado, você se depara com todos os problemas de um programa que não usasse nenhum encapsulamento.

Quando você verificar que um objeto executa mais de uma responsabilidade, precisará mover essa responsabilidade para seu próprio objeto.

Encapsulamento efetivo acontece unindo três grandes características da orientação a objetos: abstração, ocultação da implementação e divisão de responsabilidades.

Retire a abstração e você terá um código que não é reutilizável. Retire a ocultação da implementação e você ficará com um código fortemente acoplado e frágil. Retire a responsabilidade e você ficará com um código centrado nos dados, procedural, fortemente acoplado e descentralizado.

1.5 Dicas e armadilhas do encapsulamento

Dicas e armadilhas da abstração

É impossível escrever uma classe que satisfaça todos os usuários e cada situação. Não caia na fixação da abstração – resolva seus problemas primeiro, caso contrário pode-se perder nos prazos e pode criar abstrações incorretas.

A abstração deve direcioná-lo a simplificação. Se você perceber que não está simplificando o problema ao tentar abstrair você certamente está em direção errada. Não coloque em uma classe mais detalhes do que são necessários para resolver o problema.

A verdadeira abstração normalmente nasce a partir de usos reais e não do fato de um programador se sentar e decidir criar um objeto reutilizável. Normalmente os objetos reutilizáveis são derivados de um código amadurecido, que foi posto à prova e que enfrentou muitas alterações. Além de que a verdadeira capacidade de abstração também vem com a experiência.

Dicas e armadilhas do Tipo Abstrato de Dados

A transformação de um TAD em uma classe é específica da linguagem. Entretanto, existem algumas considerações independentes da linguagem que você pode fazer a respeito das classes.

A maioria das linguagens OO fornece palavras-chave que o ajudarão a definir classes encapsuladas. Primeiro, existe a própria definição de classe. A classe é como o TAD, mas com alguns recursos importantes como herança e polimorfismo que você verá no decorrer do curso.

Dentro de uma classe, normalmente você tem métodos e variáveis internos – os dados. O acesso a essas variáveis e métodos é fornecido por funções de acesso. Tudo na interface do TAD deve parecer fazer parte da interface pública do objeto.

Dicas da ocultação da implementação

Apenas os métodos que você pretende que outros usem devem estar na interface pública.

Você sempre deve ocultar as variáveis internas, a não ser que elas sejam constantes. Cremos que elas não devem estar apenas ocultas, mas também acessível apenas para a própria classe.

Não crie interfaces que apresentam apenas a representação interna com um nome diferente. A interface deve apresentar comportamentos de alto nível.

1.6 Encapsulamento e os objetivos da Orientação a Objetos

Os objetivos da programação orientada a objetos são produzir software:

1. Natural
2. Confiável

3. Reutilizável
4. Manutenível
5. Extensível
6. Oportunos

O encapsulamento atende cada um desses objetivos:

- Natural

  O encapsulamento permite dividir a responsabilidade da maneira como as pessoas pensam naturalmente. Através da abstração, você fica livre para modelar o problema em termos do próprio problema e não em termos de alguma implementação específica. A abstração permite que você pense no programa de maneira geral.

- Confiável

  Isolando a responsabilidade e ocultando a implementação, você pode validar cada componente individualmente. Quando um componente for validado, você poderá usá-lo com confiança. Isso possibilita testes de unidade completos. Você ainda precisa realizar testes de integração, para certificar-se de que o software construído funciona corretamente.

- Reutilizável

  A abstração fornece código flexível e utilizável em mais de uma situação.

- Manutenível

  O código encapsulado é mais fácil de manter. Você pode fazer qualquer alteração que queira na implementação de uma classe, sem danificar código dependente. Essas alterações podem incluir mudanças na implementação, assim como a adição de novos métodos na interface. Apenas as alterações que violam a semântica da interface exigirão mudanças no código dependente.

- Extensível

  Você pode mudar implementações sem danificar código. Como resultado, você pode fazer melhorias de desempenho e mudar funcionalidades sem danificar o código existente. Além disso, como a implementação fica oculta, o código que usar o componente será atualizado automaticamente, para tirar proveito de todos os novos recursos que você introduzir. Se você fizer tais alterações, certifique-se de fazer os testes de unidade novamente! Danificar um objeto pode ter um efeito dominó por todo o código que use o objeto.

- Oportunos

  Dividindo seu software em partes independentes, você pode dividir a tarefa de criar as partes entre vários desenvolvedores, acelerando assim o desenvolvimento.

  Uma vez que esses componentes estejam construídos e validados, eles não precisarão ser reconstruídos. Assim, o programador fica livre para reutilizar funcionalidades, sem ter de recriá-las.

1.7 Resumo

Usando encapsulamento, você pode tirar proveito das vantagens da abstração, da ocultação da implementação e da responsabilidade em seu código diário.

Com a abstração, você pode escrever objetos que são úteis em varias situações. Se você ocultar corretamente a implementação de seu objeto, estará livre para fazer quaisquer melhorias que queira em seu código – a qualquer momento. Finalmente, se você dividir corretamente a responsabilidade entre seus objetos, evitará lógica duplicada e código procedural.

Observação: Um método ou variável definido como *static* pode ser utilizado em uma classe não instanciada, ou seja, é um método ou uma variável da classe e não da instância. Outra observação é que dentro de um método *static* não se pode utilizar variáveis ou métodos internos não *static*, pois eles apenas existem quando uma instância da classe é criada e não somente da classe.

1.8 Perguntas e Respostas

1) Como você sabe quais métodos deve incluir em uma interface?

   Essa é uma tarefa simples. Pense somente nos métodos que tornarão o objeto útil, esses devem ser os métodos a serem incluídos na interface. Pense somente nos métodos necessários para que o objeto faça seu trabalho.

   Quando você começar a escrever uma interface, desejará produzir a menor interface que ainda satisfaça suas necessidades. Torne sua interface o mais simples possível. Não inclua métodos que você 'poderia' precisar. Você pode adicioná-los quando realmente precisar deles.

   Conheça certos tipos de métodos de conveniência. Se você fizer um objeto conter outros objetos, normalmente desejará evitar a criação de métodos que simplesmente encaminham uma chamada de método para um dos objetos contidos.

2) Os modificadores de acesso também têm o papel de mecanismo de segurança?

   Não. Os modificadores de acesso só restringem o modo como outros objetos podem interagir com determinado objeto. Os modificadores não têm nada a ver com a segurança do computador.

1.9 Perguntas - Exercícios

1) Como o encapsulamento atinge os objetivos da programação orientada a objetos?

2) Defina abstração e dê um exemplo demonstrando abstração.

3) Defina implementação.

4) Descreva a diferença entre interface e implementação.

5) Por que a divisão clara da responsabilidade é importante para o encapsulamento eficaz?

6) Defina TAD (Tipo Abstrato de Dados).

7) Quais são alguns dos perigos inerentes à abstração?

Ao terminar de responder as questões acima, envie-as em um arquivo texto ao seu tutor.