

```
;;=====;;
;; How to Design Programs, 2ª edição                                ;;
;; https://htdp.org/                                              ;;
;;-----;;
;; Capítulo 03: How to Design Programs                             ;;
;;-----;;
;; Resumo por: Abrantes Araújo Silva Filho                        ;;
;;          abrantesasf@gmail.com                                ;;
;;=====;;
```

No design sistemático de programas é necessário saber como, a partir da descrição de um problema, chegar em um programa correto e funcional. É necessário, por exemplo:

A) Em um nível mais genérico:

- Determinar o que é e o que não é importante
- Determinar o que o programa consumirá (inputs)
- Determinar o que o programa devolverá (outputs)
- Determinar como processar os inputs para obter os outputs

B) Em um nível mais de código:

- Saber, dada a linguagem e suas bibliotecas, quais as operações para os dados que iremos processar
- Desenvolver todas as funções necessárias (principal e auxiliares)
- Checar se o programa está correto.

Um bom programa deve estar acompanhado de uma pequena explicação sobre:

- o que ele faz
- quais os inputs necessários, com quaisquer restrições
- quais os outputs produzidos
- alguma garantia de que ele funciona e é correto

### 3.1) Design de Funções:

#### 3.1.1) Informações e Dados:

O propósito de um programa é descrever/dirigir um PROCESSO COMPUTACIONAL que consome alguma informação e produz uma nova informação. Essa informação é o domínio do programa:

- Informação: fatos a respeito do domínio do programa

O problema é que um programa não processa a informação diretamente, ela precisa ser convertida em dados compreensíveis pela linguagem de programação:

- Dados: representam a informação na linguagem de programação

Deve existir uma separação bem clara entre INFORMAÇÃO e os DADOS, entre o processo de TRADUÇÃO (parsing) e o PROCESSAMENTO (de dados):

```
Informação -> Tradução -> Dados -> Processamento -> Tradução -> Informação
                para dados                          para informação
```

O processo de TRADUÇÃO (informação para dados, ou de dados para informação) requer muita experiência de programação, portanto será visto posteriormente. O foco no momento são nos DADOS e no PROCESSAMENTO desses dados.

Dito isso, os programadores precisam primeiro definir como representar a informação como dados, e como interpretar esses dados como informações. Isso é feito através da DEFINIÇÃO DE DADOS de uma função. Essa definição indica uma classe para os dados, como criar elementos dessa classe, e como interpretar um dado nessa classe. Exemplo de definição de dados:

```
; Representação: a temperatura será um número.
; Interpretação: em graus Celsius
```

### 3.1.2) O processo de design: Receita de Design de Função (RDF)

Sabendo agora diferenciar adequadamente Informação dos Dados, um dos componentes do design sistemático de programas é trabalhar com a Receita de Design de Função (RDF). Pegue uma RDF e siga as instruções a seguir.

#### 1) DEFINIÇÃO DE DADOS:

Analise o problema, tipicamente escrito em palavras, e:

- a) Identifique a INFORMAÇÃO que deve ser representada, e defina como ela será representada pelos DADOS da linguagem de programação escolhida. Formule as DEFINIÇÕES DE DADOS. Ex.:
 

```
; Representação: a temperatura será um número
; Interpretação: em graus Celsius
```

#### 2) ASSINATURA, DECLARAÇÃO DE PROPÓSITO E CABEÇALHO DA FUNÇÃO

Para cada função a ser desenvolvida, prepare:

- a) ASSINATURA DA FUNÇÃO: é um comentário que indica quantos inputs a função necessita, quais as classes dos inputs, e o que a função produzirá. Ex.:
 

```
; Número String Imagem -> Imagem
```
- b) DECLARAÇÃO DE PROPÓSITO: é um comentário que resume o propósito da função, em uma ou poucas linhas. Deve responder claramente à pergunta: "O que essa função calculará?". Em programas com múltiplas funções, uma declaração de propósito geral do programa deve ser fornecida no começo do arquivo. Ex.:
 

```
; adiciona s à img, y pixels do topo e 10 da esquerda
```
- c) CABEÇALHO DA FUNÇÃO: é um "mockup" da função que será construída, mas já com nome definido e parâmetros (com nomes significativos) já definidos. O corpo da função pode retornar apenas um único valor da mesma classe do output desejado, pois será construído posteriormente. Ex.:
 

```
(define (add-image y s img)
  (empty-scene 100 100))
```

#### 3) EXEMPLOS FUNCIONAIS:

Ilustre a assinatura e a declaração de propósito. Prepare:

- a) EXEMPLOS FUNCIONAIS: construa alguns exemplos funcionais indicando os inputs e os outputs esperados. Escolha exemplos representativos da funcionalidade esperada da função.

#### 4) FAÇA O INVENTÁRIO E CONSTRUA UM TEMPLATE:

Faça um inventário das coisas que são necessárias para construir a função, para entender claramente quais são os inputs e o que a função precisa calcular. Prepare um template da função:

- a) TEMPLATE: substitua o corpo da função no mockup já criado e introduza os parâmetros no corpo da função, com "..." para indicar que não é uma função completa, é apenas um template que precisa ser completado.

## 5) CODIFIQUE A FUNÇÃO:

Agore é a hora de codificar a função, ou seja, programar a funcionalidade esperada, substituindo o template por código real.

## 6) TESTE A FUNÇÃO:

Teste a função com os exemplos funcionais definidos e novos casos de teste.

- a) Testes com exemplos funcionais: garantem que, no mínimo, a função produz o resultado esperado com os inputs mais básicos.
- b) Testes extensivos: planeje casos de teste que irão desafiar a função: inputs errados, inputs fora de limites, etc. Garanta que a função produza o output correto em todos os testes extensivos.

## 3.2) Quando usar a RDF?

Até que todos os passos da RDF se tornem reflexos, internalizados, use TODOS os passos e NÃO PULE nenhum. Seguir os passos garante que erros bobos sejam evitados. Até que isso seja natural, use o seguinte modelo nos arquivos:

```
;; Função: <nome da função>
;; =====
;; <declaração de propósito - 2b>
;; <assinatura - 2a>
;;   Inputs: <definição de dados - 1: representação>
;;   Output: <definição de dados - 1: interpretação>
;; Exemplos funcionais: <3>
;;   in:
;;   out:
;; Requisitos especiais:
;;   <bibliotecas, etc.>
<cabeçalho    - 2c>
<template    - 4>
<codificação - 5>

;; Testes funcionais: <6>
(funcao inputs)
```

## 3.3) Conhecimento do Domínio:

Que tipo de conhecimento é necessário para codificar o corpo de uma função? Basicamente são 3 tipos de conhecimentos de domínio:

- Domínios EXTERNOS: matemática, música, biologia, finanças, linguística, enfim: é o conhecimento do domínio do problema que a função resolverá. Frequentemente é necessário aprender com especialistas.
- Domínio da LINGUAGEM: é o conhecimento profundo, detalhado e extenso da linguagem de programação escolhida e suas bibliotecas. Conhecer a fundo mais de uma linguagem (e suas bibliotecas) lhe permitirá escolher a melhor linguagem para o problema em questão.
- Domínio de CIÊNCIA DA COMPUTAÇÃO: para dados complexos que exigem estruturas de dados complexas, é necessário o conhecimento profundo da ciência da computação.

### 3.4) De Funções aos Programas:

-----  
Em programas (batch ou interativos) que necessitem de muitas funções, mantenha uma "wish list" com as funções necessárias. Essa lista deve conter:

- nome da função
- assinatura da função
- declaração de propósito da função

O programa termina (estará pronto) quando você codificar todas as funções da wish list (ela ficar vazia).

### 3.5) Sobre os Testes:

-----  
O teste das funções (e programas) é uma atividade extremamente trabalhosa, se for feito direito. Confiar em testes manuais é ERRADO pois, como é trabalhoso, logo os programadores deixam de fazer. O correto é que, PARA CADA FUNÇÃO, seja criado junto um TESTE UNITÁRIO dessa função, com os casos de testes funcionais. Esse teste unitário garantirá que alterações futuras da função não causem um erro lógico. A linguagem BSL e o DrRacket utilizam a função CHECK-EXPECT para testes unitários, por exemplo:

- (check-expect (funcao <input>) <output\_esperado>)

Para cada exemplo funcional, crie os testes unitários!

### 3.6) Design de "World Programs":

#### 1) DEFINIÇÃO DE DADOS:

-----  
Analise o problema, tipicamente escrito em palavras, e:

- a) Identifique a INFORMAÇÃO sobre o mundo que deve ser representada, e defina como ela será representada pelos DAODS da linguagem de programação escolhida. Formula as DEFINIÇÕES DE DADOS que representam o ESTADO DO MUNDO. Ex.:
  - ; Representação: o WordlState é um número
  - ; Interpretação: o número de pixels a partir da borda esquerda

#### 2) PROPRIEDADES CONSTANTES:

Identifique as PROPRIEDADES DO MUNDO que permanecerão constantes durante o tempo, e que serão necessárias para renderizar o mundo como uma imagem (requisito da BSL). Crie CONSTANTES para cada propriedade, como um breve comentário que explica o que essas constantes significam. Divida as constantes em dois grupos:

- a) CONSTANTES FÍSICAS: atributos gerais dos objetos no mundo
- b) CONSTANTES GRÁFICAS: são as imagens dos objetos no mundo, criadas utilizando-se as propriedades físicas.

#### 3) PROPRIEDADES QUE VARIAM NO TEMPO:

Identifique as PROPRIEDADES DO MUNDO que são alteradas em reação aos EVENTOS que ocorrerão no mundo, e que se tornarão o ESTADO ATUAL do mundo. Desenvolva uma representação de dados para TODOS os possíveis estados do mundo, com suas respectivas definições de dados.

#### 4) DESIGN DAS FUNÇÕES NECESSÁRIAS:

Depois que você tem todas as definições de dados (estados do mundo) prontas, é necessário fazer o design de diversas funções (e, para cada, use uma RDF):

- a) RENDER: exibirá o estado atual do mundo ao usuário, com uma imagem.
- b) EVENT HANDLERS: serão responsáveis por monitorar a ocorrência de eventos e retornar o PRÓXIMO ESTADO do mundo. Os 3 principais event handlers são:
  - clock-tick-handler
  - keystroke-handler
  - mouse-event-handler
- c) END?: é a função responsável por decidir se o ESTADO ATUAL do mundo será atualizado para o PRÓXIMO ESTADO e, em caso afirmativo, continuar a execução do programa deixando o render exibir o estado atualizado.

#### 5) FUNÇÃO MAIN:

Com todas as funções prontas, a única coisa que falta é a função MAIN, que executará o World Program através de uma expressão big-bang. O primeiro argumento, obrigatoriamente, é o ESTADO INICIAL do mundo; o resto dos argumentos (render, event handlers, end?) podem ocorrerem em qualquer ordem.