UNIVERSIDADE FEDERAL DO CEARÁ
Campus de Quixadá

UNIVERSIDADE FEDERAL DO CEARÁ

Campus de Quixadá

Reuso de software

REFATORAÇÃO DE UMA CALCULADORA BASEADA EM PYTHON E TKINTER PELO USO DE PADRÕES DE PROJETO E MELHORES PRÁTICAS DE DESENVOLVIMENTO DE SOFTWARE

GABRIEL DIAS DE LIMA

MARCELO SILVA DE LIMA

PROF. FRANCISCO VICTOR DA SILVA PINHEIRO

QUIXADÁ, CEARÁ, NOVEMBRO DE 2025

RESUMO

Há décadas, atividades econômicas ligadas ao desenvolvimento de software vêm ganhando cada vez mais importância na sociedade contemporânea. Contudo, mesmo que existam empresas dedicadas exclusivamente a essa atividade, muitas delas ainda encontram dificuldades em produzir e utilizar ativos de software reutilizáveis, o que limita a produtividade potencial que elas poderiam alcançar. Nesse contexto, o presente projeto visa pontuar problemas comumente encontrados em ativos de software por meio da apresentação de um projeto simples de uma calculadora desenvolvida em Python em 2017 e, adicionalmente, de sua refatoração com o uso de padrões de projeto e melhores práticas de desenvolvimento, a fim de tornar seus componentes individuais mais reutilizáveis e extensíveis. Dessa forma, por meio da identificação dos code smells desse projeto e da apresentação dos seus respectivos consertos no código refatorado, busca-se compreender de forma mais aprofundada como identificar e solucionar esses problemas ao desenvolver e manter sistemas maiores e mais robustos. Ademais, para fins de documentação, ele pode ser acessado em: https://github.com/autumn-Days/projeto-refatoracao.

palavras chave: reuso de software; padrões de projeto; refatoração.

INTRODUÇÃO

Esta seção visa apresentar o sistema escolhido para análise, fornecer uma breve descrição sobre o contexto de sua criação e justificar o porquê dele se tratar de um bom arquetipo para atividades ligadas a refatoração educativa.

0.1 Escolha do projeto

O projeto selecionado para análise é uma calculadora desenvolvida em 2017 por uma equipe de três programadores, cujo código-fonte está disponível publicamente no GitHub [1]. Este projeto foi escolhido por ser altamente representativo do escopo deste trabalho, dado o elevado número de code smells identificados. Tais problemas incluem, mas não se limitam a, baixa coesão, a presença de classes monolíticas, e repetição de código, conforme será detalhado e analisado nas seções subsequentes.

Ademais, a simplicidade conceitual e a natureza concreta das aplicações de calculadora as tornam um excelente recurso didático. Dessa forma, elas podem ser facilmente apresentadas

[1] https://github.com/tasdikrahman/pyCalc/tree/master

a profissionais de diversos níveis facilmente e servem como um arquétipo eficaz para demonstrar e aplicar boas práticas de desenvolvimento de software que, posteriormente, podem ser facilmente escalados para softwares maiores e mais complexos.

0.2 Apresentação do sistema

Uma captura de tela da interface do sistema antes da refatoração é apresentada na figura 1. Como é possível observar, ela possui todas as características comuns às demais calculadoras disponíveis no mercado, visor, botões numéricos e outros meios de entrada de dados.

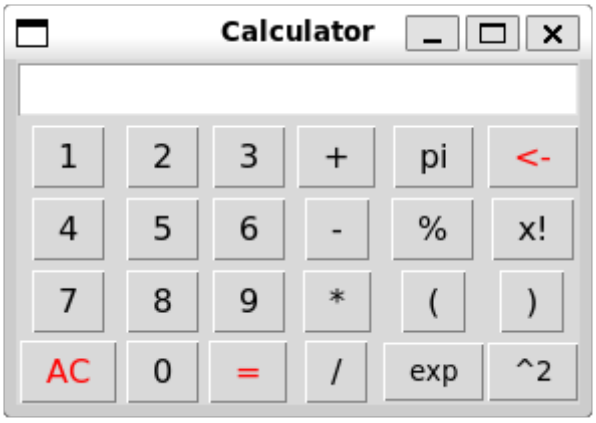


Figura 1: Captura da tela da calculadora apresentada. A estética da calculadora pode diferir entre sistemas operacionais. A tela em questão foi obtida pela execução do projeto em um Windows 11 rodando o WSL2.

Nesse contexto, ainda foi possível tornar o projeto ainda mais atrativo do ponto de vista didático ao implementar uma funcionalidade de *modo avançado*, que adicionou outras três funcionalidades ao código original, as quais foram: cálculo da raiz quadrada e cálculo em radianos do seno, cosseno e tangente.

Essa escolha foi motivada em virtude da presença dessa funcionalidade na maioria dos softwares desse domínio. Dessa forma, a figura 2 representa o resultado da incorporação do *modo avançado* ao código original.

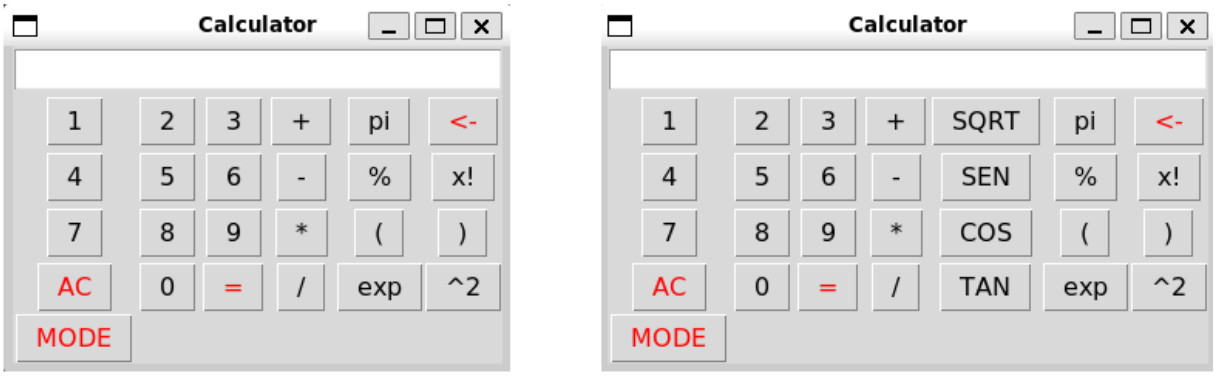


Figura 2: Captura da tela da calculadora acrescida da funcionalidade de troca de modos.

0.3 Identificação dos code smells

Os primeiros sinais de desorganização no código original vieram de seus code smells. Primeiramente, toda a criação de botões encontrava-se em uma única função, que, por sua vez, era chamada no construtor da classe principal. Tais chamadas do construtor do objeto tk.Button também se uniam de modo descriterioso, ou seja, todos os botões, independentemente do seu tipo e/ou função, encontravam-se no mesmo método. A figura 3 retrata o método responsável pela criação dos botões na interface gráfica. Além de se tratarem de repetições desnecessárias, a instanciação destes objetos também incluíam o método .grid(), utilizado para mostrar os objetos na tela, mantendo tanto a criação dos botões quanto sua exibição na tela em uma única função.

Aliado ao problema, o construtor da classe estava sobrecarregado com a criação de inúmeras variáveis e suas configurações. O construtor tinha um total de 39 linhas. Assim como o método anteriormente descrito, não haviam quaisquer separações entre os diferentes atributos do sistema, o que deixava o construtor confuso o bastante para dificultar o entendimento da responsabilidade de cada linha de código.

Outra deficiência encontrada no projeto sob análise foram as funções utilizadas para realizar as operações presentes nos botões. Todas se faziam presentes na mesma classe da interface, o que colaborava ainda mais para uma baixa coesão na classe principal, que assumia a função de uma ”classe deus”responsável por quase tudo no projeto. Por fim, essa classe era chamada na main de outro arquivo, onde o loop de execução era executado.

A calculadora encontrada possuia, portanto, problemas relativos à baixa coesão da sua classe; a aplicação de uma ”classe deus”para tratar de funções relacionadas ao que era exibido

```
def _init_ui(self):
    self.one = tk.Button(
        self, text="1", command=lambda: self.get_variables(1), font=self.FONT_LARGE)
    self.one.grid(row=2, column=0)
    self.two = tk.Button(
        self, text="2", command=lambda: self.get_variables(2), font=self.FONT_LARGE)
    self.two.grid(row=2, column=1)
    self.three = tk.Button(
        self, text="3", command=lambda: self.get_variables(3), font=self.FONT_LARGE)
    self.three.grid(row=2, column=2)

    self.four = tk.Button(
        self, text="4", command=lambda: self.get_variables(4), font=self.FONT_LARGE)
    self.four.grid(row=3, column=0)
    self.five = tk.Button(
        self, text="5", command=lambda: self.get_variables(5), font=self.FONT_LARGE)
    self.five.grid(row=3, column=1)
    self.six = tk.Button(
        self, text="6", command=lambda: self.get_variables(6), font=self.FONT_LARGE)
    self.six.grid(row=3, column=2)
```

Figura 3: Criação de botões no código original.

na tela e à lógica matemática a ser executada e code smells como código repetido e classes muito longas. Todos estes defeitos acarretariam em dificuldades para a implementação de novas funções, aumento gradual da complexidade do código, diminuição da sua compreensibilidade e problemas para manutenção do código à longo prazo.

0.4 Implementação das funcionalidades adicionais

Efetuaram-se algumas adições ao código original. Implementou-se a funcionalidade de mudança entre calculadora básica e científica ao projeto escolhido. Os botões de operações adicionais escolhidos foram o de extração da raiz quadrada e as três principais operações trigonométricas: seno, cosseno e tangente.

Neste processo, foram criados um botão MODE, para efetuar a troca; uma variável booleana para identificar o modo vigente; e um método para responsável por inverter o valor da variável e esconder ou mostrar os botões, a depender do seu valor. Esta função é passada ao construtor do botão, a fim de que ele possa executar a lógica.

Devido à estrutura do código não-refatorado, a implementação acabou por seguir os mesmos vícios do que já existia. O botão, por exemplo, foi declarado no método dedicado à criação

dos botões. O método contou com múltiplas linhas de chamadas dos métodos grid_forget() e grid(), responsáveis por esconder e mostrar o elemento na tela, respectivamente.

Outra modificação importante foi feita no parser presente na aplicação original. Ele utilizava uma biblioteca antiga, ainda do Python 2, o que dificultava a execução do programa e criava problemas constantes de compatibilidade. Ela foi substituída pelo módulo ast, que trabalha com árvores de sintaxe abstrata. A modificação se faz presente na linha 229 do código.

0.5 Implementação da refatoração

Para solucionar tais problemas, foram escolhidos quatro padrões de projeto, cada um dedicando-se a atacar um determinado problema presente no código.

Facade

```python
import front.calculatorView as front
import back.calculatorModel as calcModel
from front.state import BasicState, AdvancedState
import tkinter as tk

class CalculatorController:
    def __init__(self, model):
        self.model = model
        self.root = tk.Tk()
        self.state = BasicState(self, model)
        self.current_frame = None
        self.model.master = None

    def __destroyWidgets(self):
        for widget in self.root.winfo_children():
            widget.destroy()

    def __buildView(self):
        view_cls = getattr(self.state, "view_class", None)
        if view_cls is None:
            raise RuntimeError("O State não tem nenhum objeto 'view' associado a ele")

        frame = view_cls(self.root, self.model)
        self.model.master = frame
        frame.set_controller(self)
        frame.config_window()
        frame.config_icon()
        frame.pack()
        self.current_frame = frame

    def config_view(self):
        self.__buildView()

    def switchState(self):
        self.__destroyWidgets()
        self.state.switch()
        self.__buildView()

    def run_app(self):
        self.config_view()
        self.root.mainloop()

```

Figura 4: Métodos da classe CalculatorFacade do código refatorado.

O padrão Facade foi utilizado para tratar de um problema decorrente da reestruturação das classes e do construtor da classe gráfica (CalculatorView). Devido à reestruturação do código em várias classes e funções auxiliares, a main precisava executar mais funções do que originalmente, além de precisar tratar cada parâmetro a ser importado pelos objetos. A Figura 4

retrata parte do código da classe Facade.

Por essa razão, implementou-se uma solução baseada no padrão Facade, representada pela classe CalculatorFacade. Nela abstrai-se a criação e a passagem de argumentos entre as diferentes classes do sistema, permitindo que a função main seja enxuta, contendo apenas a criação do objeto CalculatorFacade e a execução do loop principal do programa. Além disso, a Facade também é responsável pela declaração inicial do estado relacionado ao padrão State, que será explicado posteriormente.

Factory

```
class FabricaBotoes:
    def create(self, button:str):
        if button.isdigit():
            return self.__criarBotaoDigito(button)
        elif button in self.operators:
            return self.__criarBotaoOperador(button)
        elif button == "MODE" or button == "SQRT":
            label,callback,cor = self.token2parameters[button]
            return self.__criarBotaoCustomizavel(button, callback, cor)
        else:
            if self.__isAngle(button):
                methodName = self.angles2methods[button]
                callback_callable = (lambda m=methodName: getattr(self.model, m)())
                return Botao(button, self.FONT_LARGE, callback_callable, "black", master=self.master)

            key = button
            if self.__isAngle(button):
                key = "ANG"
            label,callback,color = self.token2parameters[key]
            return self.__criarBotaoEspecial(label,callback,color)

    def __criarBotaoOperador(self, button:str):···

    def __criarBotaoDigito(self, button:str):···

    def __criarBotaoEspecial(self, label:str, callback:str, cor:str):···

    def __criarBotaoCustomizavel(self, label, callback, cor):···

    def __isAngle(self,angle):···
```

```
from models.button import Botao
from back.calculatorModel import *
from models.builder import BuilderCalc

class FabricaBotoes:
    def __init__(self, master, model):···

    def create(self, button:str):
        if button.isdigit():
            return self.builder.buildDigit(button)
        elif button in self.operators:
            return self.builder.buildOperator(button)
        elif button == "MODE" or button == "SQRT":
            label,callback,cor = self.token2parameters[button]
            return self.builder.buildCustom(button, callback, cor)
        else:
            if self.__isAngle(button):
                callbackName = self.angles2methods[button]
                return self.builder.buildAngle(button,callbackName,"black")
            else:
                label,callback,color = self.token2parameters[button]
                return self.builder.buildSpecial(label,callback,color)
    def __isAngle(self,angle):
        return angle in set(["SEN","COS","TAN"])
```

Figura 5: A figura apresenta uma comparação entre o padrão *Factory* antes e depois da implementação do *Builder* como um padrão auxiliar à legibilidade e a manutenibilidade do módulo *FactoryButtons*.

Primeiramente, viu-se potencial no uso do Factory para a criação dos botões, uma vez que, no modo que era feita, tal tarefa resultava em código repetido e métodos muito extensos. O Factory permite a criação de objetos com características similares, mas com pequenas nuances de um para o outro, tornando-o ideal para o cenário escolhido. A chamada do construtor Botao a classe *Botao* foi substituída pela criação de um objeto FabricaBotoes responsável por receber o tipo de um botão e atribuí-lo ao seu label e função à ser executada.

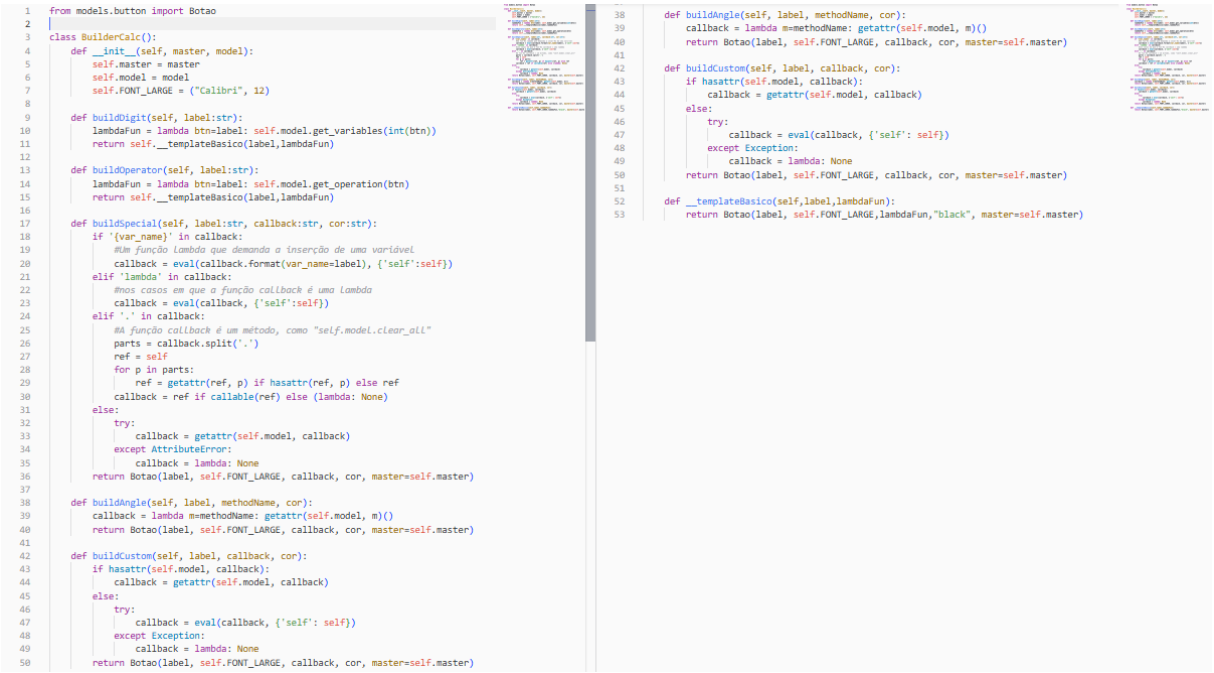
O *FabricaBotoes* permite a criação de três tipos de botão: botões de digito, ou seja, numéricos; botões de operação, como adição, subtração e multiplicação; e botões especiais, que necessitam de métodos específicos para suas funcionalidades. Há também a possibilidade de criar um botão MODE, responsável pela troca da calculadora básica para a avançada. O método .create() recebe uma string com o modo do botão a ser criado e retorna um objeto da

classe Botao, que, por sua vez, é recebido pela classe responsável por sua exibição.

Builder

Após a implementação da classe *FabricaBotoes* foi notado que ela havia ficado demasiadamente grande e com vários métodos privados que poderiam dificultar a manutenção do módulo por meio de conflitos de *merge*, por exemplo. Nesse contexto, uma solução apropriada para esse problema foi a implementação do padrão de *builder*, o que viabilizou a resolução do problema ligados a classes grandes. A imagem seguinte apresenta o antes o depois da mudança.

Nesse contexto, o objetivo principal foi mover a lógica dos métodos privados da classe *FabricaBotoes* para a classe *Builder*, que se encarrega da construção dos componentes da interface gráfica. A figura seguinte evidencia o funcionamento do *Builder*.



```python
from models.button import Botao

class BuilderCalc():
    def __init__(self, master, model):
        self.master = master
        self.model = model
        self.FONT_LARGE = ("Calibri", 12)

    def buildDigit(self, label:str):
        lambdaFun = lambda btn=label: self.model.get_variables(int(btn))
        return self.__templateBasico(label,lambdaFun)

    def buildOperator(self, label:str):
        lambdaFun = lambda btn=label: self.model.get_operation(btn)
        return self.__templateBasico(label,lambdaFun)

    def buildSpecial(self, label:str, callback:str, cor:str):
        if '{var_name}' in callback:
            #Um função lambda que demanda a inserção de uma variável
            callback = eval(callback.format(var_name=label), {'self':self})
        elif 'lambda' in callback:
            #nos casos em que a função callback é uma lambda
            callback = eval(callback, {'self':self})
        elif '.' in callback:
            #A função callback é um método, como "self.model.clear_all"
            parts = callback.split('.')
            ref = self
            for p in parts:
                ref = getattr(ref, p) if hasattr(ref, p) else ref
            callback = ref if callable(ref) else (lambda: None)
        else:
            try:
                callback = getattr(self.model, callback)
            except AttributeError:
                callback = lambda: None
        return Botao(label, self.FONT_LARGE, callback, cor, master=self.master)

    def buildAngle(self, label, methodName, cor):
        callback = lambda m=methodName: getattr(self.model, m)()
        return Botao(label, self.FONT_LARGE, callback, cor, master=self.master)

    def buildCustom(self, label, callback, cor):
        if hasattr(self.model, callback):
            callback = getattr(self.model, callback)
        else:
            try:
                callback = eval(callback, {'self': self})
            except Exception:
                callback = lambda: None
        return Botao(label, self.FONT_LARGE, callback, cor, master=self.master)
```

```python
    def buildAngle(self, label, methodName, cor):
        callback = lambda m=methodName: getattr(self.model, m)()
        return Botao(label, self.FONT_LARGE, callback, cor, master=self.master)

    def buildCustom(self, label, callback, cor):
        if hasattr(self.model, callback):
            callback = getattr(self.model, callback)
        else:
            try:
                callback = eval(callback, {'self': self})
            except Exception:
                callback = lambda: None
        return Botao(label, self.FONT_LARGE, callback, cor, master=self.master)

    def __templateBasico(self,label,lambdaFun):
        return Botao(label, self.FONT_LARGE,lambdaFun,"black", master=self.master)
```

Figura 6: Conteúdo do módulo ”Builder”.

State

Por último, o State foi implementado para atacar a mudança de modo entre a calculadora básica e a científica. Primeiramente, criou-se uma classe interface CalculatorState, que serviria de base para as duas classes herdadas, BasicState e AdvancedState. Ambas recebem outras duas classes oriundas do view, CalculatorBasic e CalculatorAdvanced, respectivamente. Tais classes do view contém os atributos de cada estado, ou seja, os botões específicos de cada comportamento.

```python
class AdvancedState(CalculatorState):
    view_class = view.CalculatorAdvanced

    def switch(self):
        from front.state.basic_state import BasicState
        self.controller.state = BasicState(self.controller, self.model)

class BasicState(CalculatorState):
    view_class = view.CalculatorBasic

    def switch(self):
        from front.state.advanced_state import AdvancedState
        self.controller.state = AdvancedState(self.controller, self.model)
```

Figura 7: Classes de estado presentes no código finalizado.

Além disso, as classes BasicState e AdvancedState possuem um método switch(), que é responsável por realizar a troca de estado. Em ambas, a classe do objeto é trocada para a oposta. Assim, quando o botão MODE na interface é pressionado, a função executa a inversão de estado sem a necessidade de uma estrutura de if/else e booleanos. A Figura 7 mostra a implementação de ambas as classes.

CONCLUSÃO

Concluído o processo, pode-se notar de imediato diversos benefícios resultantes do uso dos padrões de projeto. Primeiramente, devido à "quebra" do código em classes e métodos menores, o primeiro ponto positivo foi um aumento significativo na coesão das classes. Como a calculadora antiga foi implementada por uma única classe responsável pelo que era exibido na tela e pela lógica de programação, a separação do código em objetos menores resultou em diversas classes com responsabilidades bem definidas.

Alta coesão também significa maior manutenibilidade e facilidade ao implementar novas funções, o que pôde ser notado após a adição da funcionalidade da calculadora avançada. No código original, no qual foi modificado o mínimo possível, a forma encontrada para criar os novos botões foi análoga à presente no código, ou seja, a criação de diversos botões no método _init_ui(). Para alternar entre os modos, a solução foi declarar diversas chamadas das funções "grid_forget()" e "grid()" para, respectivamente, esconder e mostrar os botões adicionais. No código refatorado, todavia, bastou adicionar poucos parâmetros ao State para per-

mitir a alternância entre os tipos de calculadora, enquanto a criação dos botões foi facilitada pela implementação do Factory. Todos esses fatores reduziram significativamente a repetição de código encontrada, o que comprova os benefícios da refatoração para a manutenção, a implementação de novas funcionalidades no código e a minimização da repetição de código.

Outrossim, é interessante notar que, mesmo com os benefícios evidentes, o processo de refatoração introduziu novos desafios de *design*. A criação e organização de novas classes, embora tenha elevado a coesão, resultou em um crescimento na complexidade estrutural do código. Esse aspecto pode introduzir uma curva de aprendizado mais elevada para pessoas novas ao projeto.

No que tange ao acoplamento, foi observado que as classes exibem algum grau de dependência, o que se deve em grande medida as exigências do pacote gráfico utilizado. O framework Tk impõe restrições arquitetônicas que demandam o acesso direto de algumas classes a outras para o funcionamento pleno da aplicação.

No entanto, mesmo com essas novas características, acredita-se que a refatoração foi majoritariamente positiva para o projeto. As modificações facilitam a implementação de novas funcionalidades, permitem uma manutenibilidade maior e possibilitam que, em caso de bugs ou comportamentos indesejados, o código problemático seja corrigido sem afetar o restante da calculadora.