PRN calculator

Jay

|  |  |  |  |
| --- | --- | --- | --- |
| version | auther | TIME | DESCRIPTION |
| 1.0 | Jay | 23th SEPT | Create document |
| 2.0 | Jay | 25th SEPT | Add session to explain how to extend more commands  Upgrade some sample |
|  |  |  |  |

# System functions

This is a PRN calculator. You can use this application to do calculations. As we do not design any graphic interface, the only tool you can use is the console. In this application , we have command line to support some operations of application and command to support math operations.

## Command line

|  |  |
| --- | --- |
| Command line | description |
| “man” | This is the “manuals” command. It will show you the usage of command |
| “exit” | This is a special command “exit”. Exit calculator application |
| Calculator command line | all the prn calculator logic will be support by calculator command line |

Sample for man

|  |
| --- |
| man  command:'\*'. sample: 4\*4=16, you need to input '4 4 \*' and out put wii be '16'  command:'+' sample: 1+2=3, you need to input '1 2 +' and out put wii be '3'  command:'-'. sample: 4-1=3, you need to input '4 1 -' and out put wii be '3'  command:'clear' sample: clear all the element in stack  commandLine:'exit'. exit application  command:'undo'. undo the last operate you did  command:'/' sample: 8/4=3, you need to input '8 4 /' and out put will be '2'  commandLine:'man'. show system commandline and command  command:'sqrt'. sample: 4 sqrt=2, you need to input '4 sqrt' and out put wii be '2'  stack: |

## Command

|  |  |
| --- | --- |
| Command | Description |
| + | 1 1 +  Stack:2 |
| - | 1 1 –  Stack:0 |
| \* | 2 1 \*  Stack:2 |
| / | 4 2 /  Stack:2 |
| “undo” | Undo the last operation |
| “clear” | Clear all the numbers in stack |

## Other special logics

### About input and display

1. Deal with numbers , set precise of input 15, output 10.

For example:

|  |
| --- |
| 1.12345678901234567  stack:1.1234567890  in this case we save “1.123456789012345” into stack and print “1. 1234567890” |

### About division

1. Deal with the precision in division, set the precise of result 15 division to 15 to save. And remove all the zero from right side.

For example:

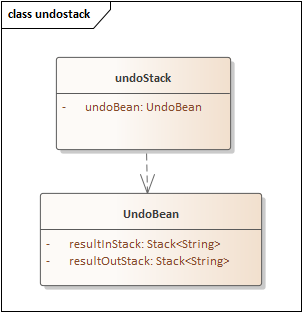
|  |
| --- |
| Case1:  4 3 /  stack:1.333333333333333  In this case if we do not set scale, bigdecimal.divide(x) will throw exception.  Case2:  4 2 /  stack:2  In this case if we set scale to 15, the result will be 2.000000000000000. so we need to remove zero from right side. |

1. Deal with denominator. We do not allow denominator to be zero, so when the denominator is zero, it will throw exception.

|  |
| --- |
| 2 0 /  denominator is zero which is not allowed in divide operation  stack: |

### About undo

I created another undo stack to save the operation result. Once there is an operation happens, there might be some data which are moved out from stack, and those number(s) will be set into “undoBean.resultOutStack”; meanwhile there will be one (or no) data which is set into stack, and that will be set into “undoBean.resultInStack”, and “undoBean” will be set into “undoStack”. So the undo stack will record all the operation happened. The only thing which will happen when “undo” is called is to recover. Get latest “undoBean”, get number(s) from “resultInStack” and remove them from stack by order and get number(s) from “resultOutStack” and set back to the stack by order.



### About rounding mode

I use Rounding.Down as the rounding mode.

### About errors

|  |
| --- |
| *ERROR\_COMMANDLINE\_EMPTY*("ERROR\_COMMANDLINE\_EMPTY"), *ERROR\_INSUFFICIENT\_PARAM*("ERROR\_INSUFFICIENT\_PARAM"), *ERROR\_PARAM\_IS\_NOT\_NUMBER*("ERROR\_PARAM\_IS\_NOT\_NUMBER"), *ERROR\_UNDO\_INFO\_MISMATCH*("ERROR\_UNDO\_INFO\_MISMATCH"), *ERROR\_DENOMINATOR\_IS\_ZERO*("ERROR\_DENOMINATOR\_IS\_ZERO"); |

#### Business error

The codes below can be teste.

*ERROR\_INSUFFICIENT\_PARAM*

|  |
| --- |
| /  operator </> (position:0): insufficient parameters  stack: |

*ERROR\_PARAM\_IS\_NOT\_NUMBER*

|  |
| --- |
| xzy  current input is not number! input is:[xzy]  stack: |

*ERROR\_DENOMINATOR\_IS\_ZERO*

|  |
| --- |
| 2 0 /  denominator is zero which is not allowed in divide operation  stack: |

#### Internal error

The codes below are internally defined which can hardly be tested in normal case.

*ERROR\_COMMANDLINE\_EMPTY*

I will internally check whether the command line is empty, if yes system will throw this exception.

*ERROR\_UNDO\_INFO\_MISMATCH*

I will check element in undoBean and stack if they are not match I will throw this exception.

# Result of delivery

## What do I deliver

1. PRN document (this document)

It introduces the system functions, result of delivery, system structure etc.

1. Java project

I push the code to GitHub so anyone can access to the project.

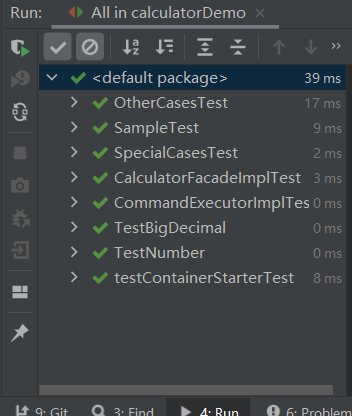
1. Runnable Jar

You can run it using “java -jar calculatorDemo-2.0-release-jar-with-

dependencies.jar”

## About the test

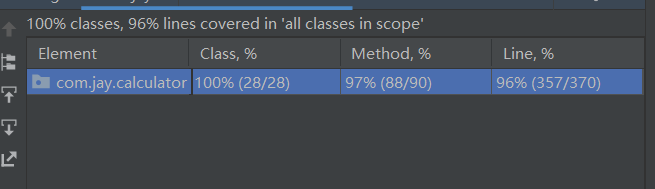
Test cases



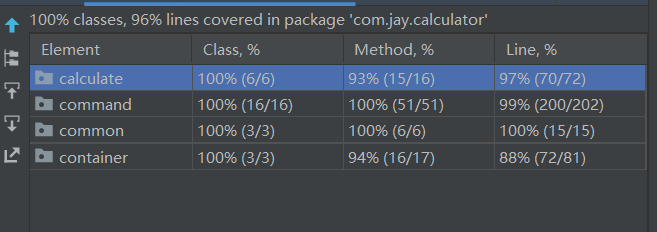
The major three TestCase classes are below.



Test coverage



Detail coverage info



What I did not cover why do not cover it？

1. CommandLineExit

This is the exit command line ,if I cover it in test case, thread will exit.

1. CalculateCommandBase.takeOutResult

There is an internal exception which will throw when system error and that is not the case we can run, it is just system robustness thinking.

1. FilePathClassLoader.getFileListInJar

This is a part of container. It will scan file list in application. Since we will package the application as a runnable jar, this method will support 2 ways. 1 is scan file out of jar; 2 is scan file in jar. Unfortunately the test case just run in Ide, which do not have jar cases. So I did not cover it.

## Todo

As a production system, we might think about how to find production issues, process production emergency issues.

1. About logs.

As it is a very small project, in order to reduce the dependencies, I did not add some logs to do the monitoring or re-play. But if it is a production server-side application that is needed.

1. About soft rollback

In order to reduce the impact scope when we find some production issue, it should have some soft toggles for some important functions. once there are some issues, we can close the function using those toggles.

# How to extend

As a calculator it might probably have some requirement to add more operators, then how can we add one more operator?

## Extend a new operator

1. Add one command

|  |
| --- |
| @Service()  @ManInfo(usage = "command: xx ,this is the useage of current command")  public class CommandXx implements CalculateCommand {  @Override  public void processCommand() throws ServiceException {  // do something here  }  } |

1. You need to write a command implement interface CalculateCommand
2. You should add annotation @Service on the class so that my container will know it should create instance in applicationContext.
3. You should add annotation @ManInfo on the class and write the usage of this command so that commandLine “man” can show how to use this command.
4. The most important thing is to implement ”processCommand()” method.
5. Define enum in OperatorCommandEnum

|  |
| --- |
| public enum OperatorCommandEnum {  /\*\*  \* you can register command here once you have new operator and implement the logic with command  \*/  XX("xx", CommandXx.class),  ADD("+", CommandAdd.class),  MINUS("-", CommandMinus.class),  TIMES("\*", CommandTimes.class),  DIVIDE("/", CommandDivide.class),  CLEAR("clear", CommandClear.class),  SQRT("sqrt", CommandSqrt.class),  UNDO("undo", CommandUndo.class);  private String operator;  private Class cls;  OperatorCommandEnum(String operator, Class cls) {  this.operator = operator;  this.cls = cls;  }  。。。。  } |
|  |

1. You need to define a new enum in this class. There are 2 fields in this enum , operator and class. Operator must match what is input.

## Extend a new command line

Please check the sample of command line “exit”.

1. Add one commandLine

|  |
| --- |
| @Service()  @ManInfo(usage = "commandLine:'exit'. exit application")  public class CommandLineExit implements CommandLine {  @Override  public void doCommand(String commandLine) {  System.exit(0);  }  } |

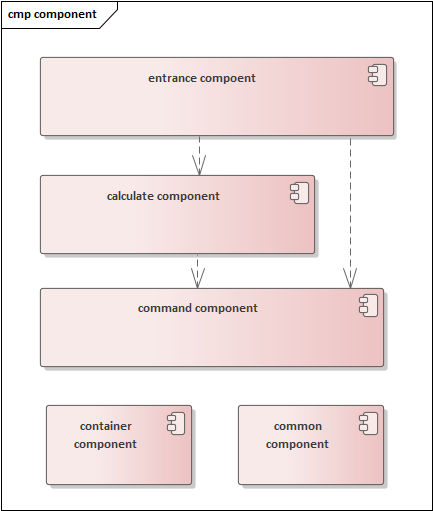
1. Add in enum

|  |
| --- |
| public enum CommandLineEnum {  /\*\*  \* you can register command here once you have new operator and implement the logic with command  \*/  CALCULATE("CALCULATE", CommandLineCalculate.class),  EXIT("exit", CommandLineExit.class),  MAN("man", CommandLineMan.class);  private String commandLine;  private Class cls;  CommandLineEnum(String commandLine, Class cls) {  this.commandLine = commandLine;  this.cls = cls;  }  。。。  } |

# System structure

## Components

There are five components in this application：Entrance component, Calculate component, Command component, Container component, Common component



### Entrance component

It is the entrance of system. Start system engine , get input call calculate component, get and print result.

### Calculate component

It is will deal with command line. If the command line belongs to calculator command then call command component.

### Command component

It is the core engine of this application, deal with the math operation and operate data in cache.

### Container component

It is a container to manage instance. In order to reduce dependencies I choose to write the container by myself, hope that will not cause confusing from you.

### Common component

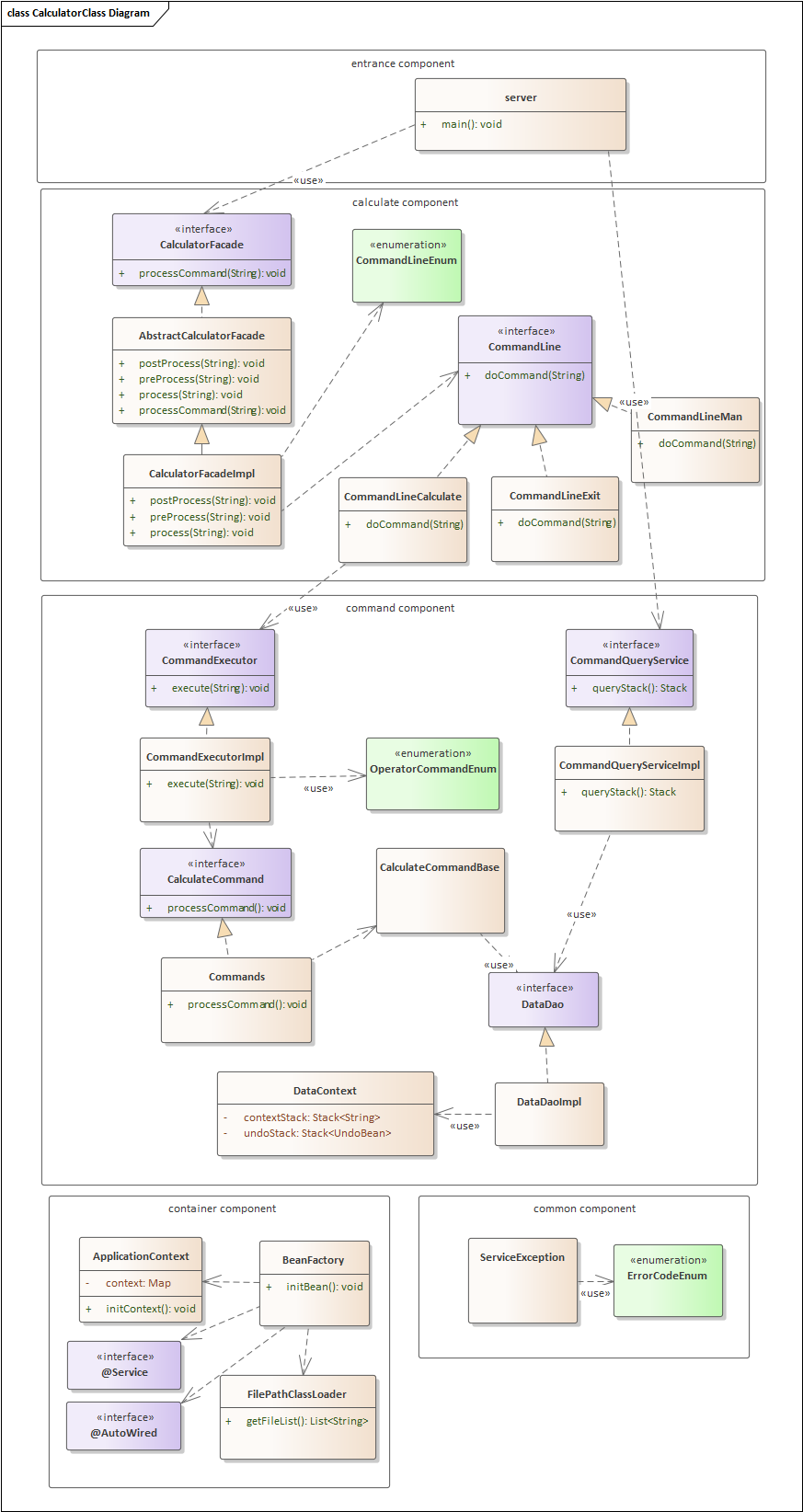
I just put exception and error code in this component.

## Classes

The class diagram see below. The dependencies are from top to bottom.

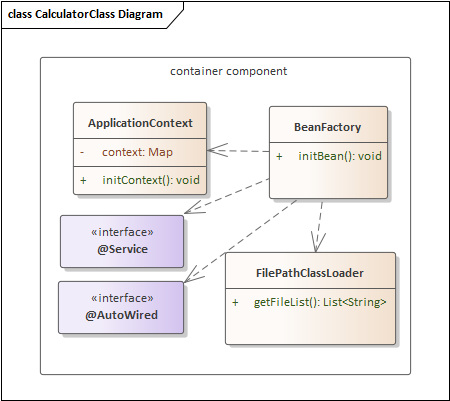
Each component has some endpoint for the other component to use.

Container is a



### Detail introduction

#### Container component

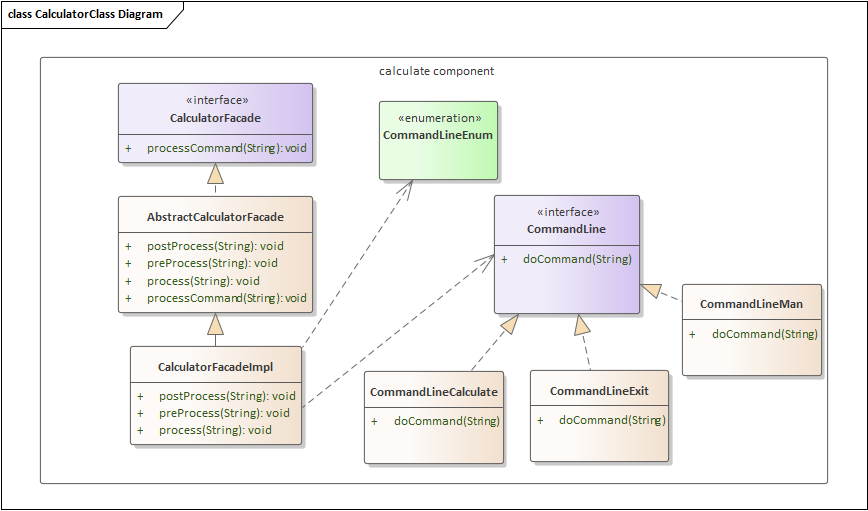


It scans file system both in jar and out of jar.

The container will do 2 things below

1. Find class with @Service annotation to create instance and put into ApplicationContext.
2. Find fields which are already in context and have some @AutoWired annotations on fields. once there are some fields match the condition container will set reference bean into those fields.

#### Calculate component:



design pattern: template, command

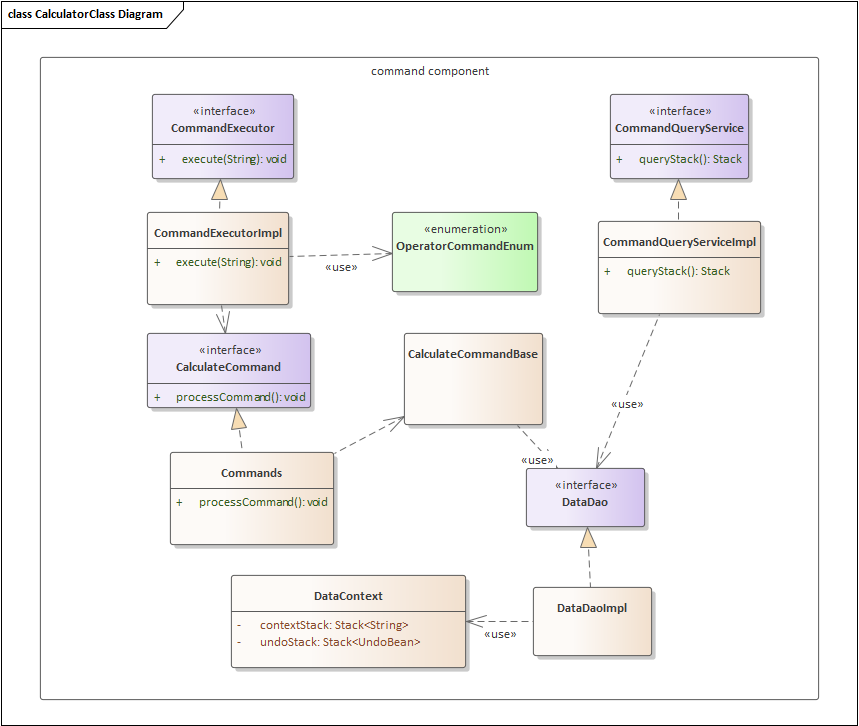
AbstractCalculatorFacade is a template which define the overall process.

CalculatorFacadeImpl is the implement of template, focus on process method. It will find out the right CommandLine and call that.

CommandLine is an interface. The detail command implement that will deal with the commandline logic.

CommandLineCalculate will call command component

#### Command component:



There are 2 interface in this component

CommandExecutor is the interface of calculate work flow . CommandExecutorImpl implements that ,identify the right command and call command to process request.

Command module has CalculateCommand as the interface , CalculateCommandBase as a role of service contains most method deal with data using dao , And detail command will deal with the math method.

Dal is the data access layer. There is an interface, an implement and a context(cash) inside.

CommandQueryService is another endpoint who provide inquiry function.