UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS214

Practical 4 Specifications

Release Date: 03-10-2025 at 23:59

Due Date: 10-03-2025 at 23:59

Total Marks: XX

Read the entire specification before starting with the practical.

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you begin coding.***
- This assignment should be completed individually.
- **Every submission will be inspected with the help of dedicated plagiarism detection software.**
- Be ready to upload your assignment well before the deadline.
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or structure).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements.**Please ensure you use C++11**
- The usage of ChatGPT and other AI-Related software to generate submitted code is strictly forbidden and will be considered as plagiarism.

2 Overview

In this practical, you will be exposed to several Gang of Four (GoF) design patterns. You will learn how to structure objects hierarchically, encapsulate ticket operations as objects, and delegate ticket handling responsibilities across different handlers. Also, you will be able to construct a complex ticket category structure step by step.

This practical will strengthen your understanding of object-oriented design principles, while giving you hands-on experience in combining multiple patterns in a single system.

3 Background

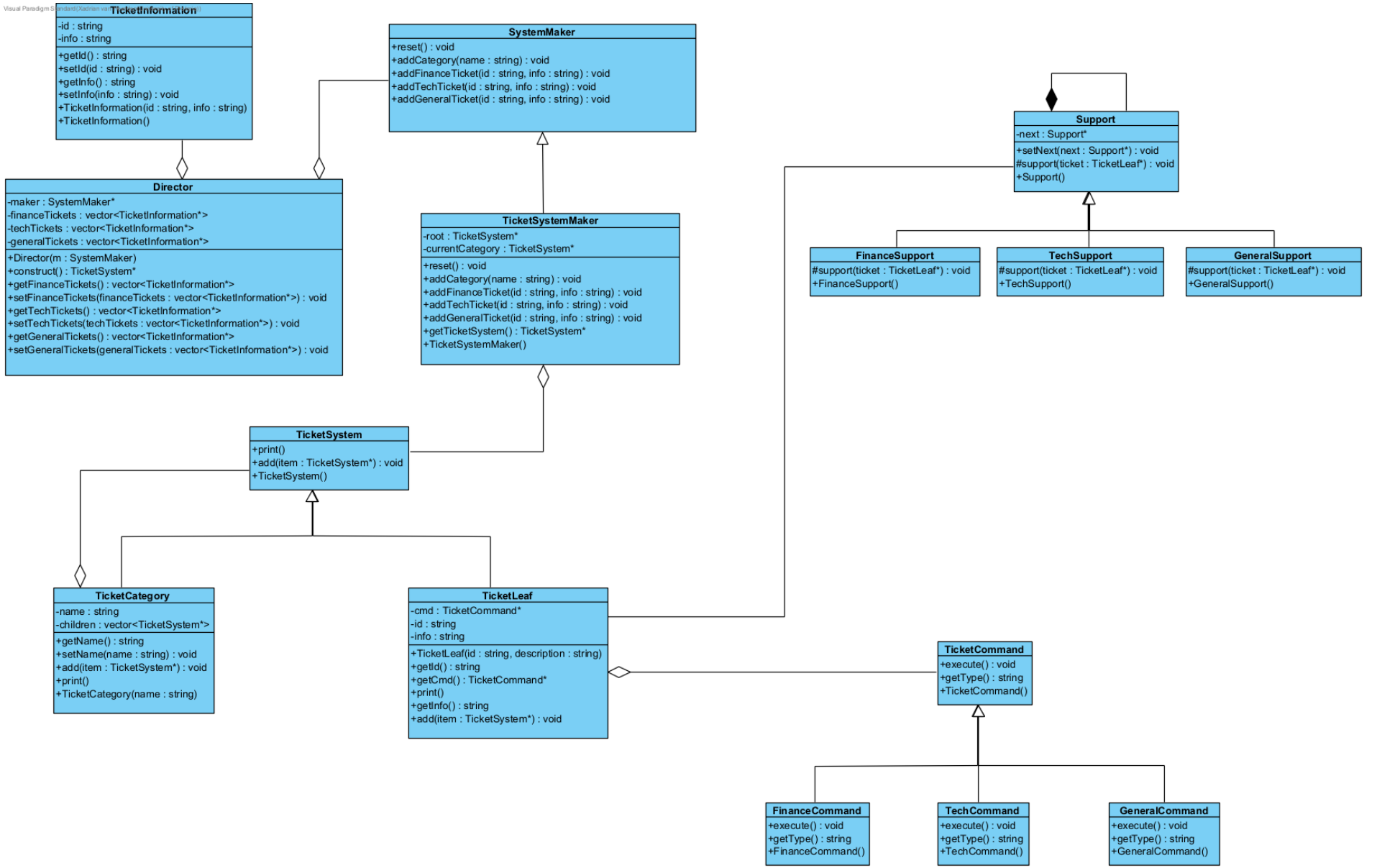
In modern organisations, customer support and internal issue tracking systems play a crucial role in ensuring smooth operations. A ticket management system is often used to log, categorise, and resolve requests such as technical issues, financial queries, or general inquiries. These systems can become very complex as they need to:

- Organise tickets into categories and subcategories for easier management.
- Encapsulate ticket actions (such as processing a refund, resetting a password, or answering a query) so that they can be executed flexibly and extended in the future.
- Route tickets to the correct support department automatically, reducing the chance of mismanagement.
- Construct complex ticket structures (categories containing multiple tickets and subcategories) without forcing clients of the system to know the low-level construction details.

This is where software design patterns are not just useful but necessary. Such a system could be used in real-world customer support platforms (like Zendesk, Freshdesk, or internal IT helpdesks), where tickets flow through multiple stages of processing and need to be structured, executed, and delegated cleanly.

Task	Marks
1_Commands	X
2_TicketSystem	X
3_SystemMaker	X
4_HelpDesk	X
Testing	X

Table 1: Mark allocation


TicketInformation
-id : string
-info : string
+getId() : string
+setId(id : string) : void
+getInfo() : string
+setInfo(info : string) : void
+TicketInformation(id : string, info : string)
+TicketInformation()
SystemMaker
+reset() : void
+addCategory(name : string) : void
+addFinanceTicket(id : string, info : string) : void
+addTechTicket(id : string, info : string) : void
+addGeneralTicket(id : string, info : string) : void
Director
-maker : SystemMaker*
-financeTickets : vector<TicketInformation*>
-techTickets : vector<TicketInformation*>
-generalTickets : vector<TicketInformation*>
+Director(m : SystemMaker)
+construct() : TicketSystem*
+getFinanceTickets() : vector<TicketInformation*>
+setFinanceTickets(financeTickets : vector<TicketInformation*>) : void
+getTechTickets() : vector<TicketInformation*>
+setTechTickets(techTickets : vector<TicketInformation*>) : void
+getGeneralTickets() : vector<TicketInformation*>
+setGeneralTickets(generalTickets : vector<TicketInformation*>) : void
TicketSystemMaker
-root : TicketSystem*
-currentCategory : TicketSystem*
+reset() : void
+addCategory(name : string) : void
+addFinanceTicket(id : string, info : string) : void
+addTechTicket(id : string, info : string) : void
+addGeneralTicket(id : string, info : string) : void
+getTicketSystem() : TicketSystem*
+TicketSystemMaker()
Support
-next : Support*
+setNext(next : Support*) : void
#support(ticket : TicketLeaf*) : void
+Support()
FinanceSupport
#support(ticket : TicketLeaf*) : void
+FinanceSupport()
TechSupport
#support(ticket : TicketLeaf*) : void
+TechSupport()
GeneralSupport
#support(ticket : TicketLeaf*) : void
+GeneralSupport()
TicketSystem
+print()
+add(item : TicketSystem*) : void
+TicketSystem()
TicketCategory
-name : string
-children : vector<TicketSystem*>
+getName() : string
+setName(name : string) : void
+add(item : TicketSystem*) : void
+print()
+TicketCategory(name : string)
TicketLeaf
-cmd : TicketCommand*
-id : string
-info : string
+TicketLeaf(id : string, description : string)
+getId() : string
+getCmd() : TicketCommand*
+print()
+getInfo() : string
+add(item : TicketSystem*) : void
TicketCommand
+execute() : void
+getType() : string
+TicketCommand()
FinanceCommand
+execute() : void
+getType() : string
+FinanceCommand()
TechCommand
+execute() : void
+getType() : string
+TechCommand()
GeneralCommand
+execute() : void
+getType() : string
+GeneralCommand()

4 Classes

Implement the UML diagram and functions as described on the following pages.
Note that some classes might not explicitly mention the use of a constructor or destructor; thus, you can assume the default constructor will be used, but with the destructors, you might have to implement the deletion of some pointers (dynamically allocated memory).

4.1 TicketInformation

This class is used to store basic information about a ticket.

- members:
  - `id:string`
    The unique identifier of the ticket.
  - `info:string`
    The description or details of the ticket.
- functions:
  - `TicketInformation()`
    The default constructor.
  - `TicketInformation(id: string, info: string)`
    Constructor. Initialises the ticket information with the given ID and description.
  - `setId(id: string): void`
    Sets the ID of the ticket.
  - `setInfo(info: string): void`
    Sets the description or details of the ticket.
  - `getId(): string`
    Returns the ID of the ticket.
  - `getInfo(): string`
    Returns the description or details of the ticket.

4.2 TicketCommand

This is the abstract base class for all commands that tickets execute.

- functions:
  - `TicketCommand()`
    The default constructor
  - `execute(id: string, info: string): void`
    Pure virtual function that executes the command logic.
  - `getType(): string`
    Returns the type of the command

4.3 FinanceCommand, TechCommand, GeneralCommand

These classes inherit from `TicketCommand` and implement ticket-specific behaviour.

- functions:
  - `FinanceCommand()` The default constructor. (the rest will also have default constructors)
  - `execute(id: string, info: string): void`
    This function should output in the terminal based on the type of command.
    Note:
    1. _ represents a space
    2. <..> refers to some information that should be brought into the output
    3. `<newline>` is a newline operator

    Output:
    - FinanceCommand:

```
Finance_Code:_<id><newLine>
Finance_Information:_<info><newLine>
```

    - TechCommand:

```
Tech_Issue:_<id><newLine>
Tech_Information:_<info><newLine>
```

    - GeneralCommand:

```
General_Number:_<id><newLine>
General_Information:_<info><newLine>
```

  - `getType(): string`
    Returns `"Finance"`, `"Tech"`, or `"General"` respectively.

4.4 TicketSystem

This is the abstract base class for the ticket system.

- functions:
  - `TicketSystem()`
    Default constructor
  - `~ TicketSystem()`
    Default destructor
  - `print(): void`
    Pure virtual function to print ticket/category details.
  - `add(item: TicketSystem*): void`
    Pure virtual function to add a component to the ticket system

4.5 TicketCategory

This class represents a category that can contain other categories or tickets

- members:
  - `name: string`
    The name of the category.
  - `children: vector<TicketSystem*>`
    The list (saved as a vector) of child components.
- functions:
  - `TicketCategory(name: string)`
    A constructor that assigns the name of the category and initializes an empty vector.
  - `getName(): string`
    Returns the category name.
  - `setName(name: string): void`
    Sets the category name.
  - `add(item: TicketSystem*): void`
    Adds a ticket or subcategory in the childrens vector.
  - `print(): void`
    Prints the category and all child elements. Printing of a category will follow the following structure:

    ```
    [<Category1Name>:_(<1stChild.print>,<2ndChild.print)]
    ```

    Note:
    1. _ represents a space
    2. <..> refers to some information that should be brought into the output
    3. <newline> is a newline operator

4.6 TicketLeaf

This class represents an individual ticket

- members:
  - `cmd: TicketCommand*`
    The command object encapsulating ticket logic.
  - `id: string`
    The ticket identifier.
  - `info: string`
    The description or details of the ticket.
- functions:
  - `TicketLeaf(id: string, info: string)`
    Constructor that sets the id of the ticket and the details of the ticket, and sets the TicketCommand to NULL.
  - `getId(): string`
    Returns the ID of the ticket.
  - `getInfo(): string`
    Returns the description or details of the ticket.
  - `getCmd(): TicketCommand*`
    Returns the associated command.
  - `setCmd(cmd:TicketCommand*):void`
    Sets the associated command.
  - `add(item:TicketSystem*):void`
    The add logic for the leaf node, it should be stubbed ;)
  - `print(): void`
    Prints the ticket details in the following format:

    ```
    <TicketID>_-_<TicketInfo>
    ```

    Note:
    1. _ represents a space
    2. <..> refers to some information that should be brought into the output
    3. <newline> is a newline operator

Note the following example about the categories and tickets

If there is a category named "AllTickets" that has two children, both of which are categories with the names "Tech" and "General", respectively. Tech has two tickets as children with the following details

1. id = "T01" and Info = "Test"

2. id = "T02" and Info = "test2"

General has one ticket with the following details:

1. id = "G01" and Info = "Gen1"

It will print the following structure:

```
[AllTickets:_([Tech:_(T01_-_Test,T02_-_test2)],[General:_(G01_-_Gen1)])]
```

Note:

1. _ represents a space
2. <..> refers to some information that should be brought into the output
3. <newline> is a newline operator

4.7 Support

This is the abstract base class for the Support chain. Ensure the constructor sets next to null on initialization

- members:
  - `next: Support*`
    The next handler in the support chain.
- functions:
  - `Support()`
    The default constructor should assign NULL to the next member variable
  - Here the destructor plays an important role ;)
  - `setNext(next: Support*): void`
    Sets the next handler in the chain.
  - `support(ticket: TicketLeaf*): void`
    Tries to handle the ticket, or passes it to the next handler.

Note: Don't forget about virtual functions

4.8 FinanceSupport, TechSupport, GeneralSupport

These classes extend Support and handle their respective ticket types.

- functions:
  - Default constructors

– `support(ticket: TicketLeaf*): void`

Checks if the ticket is of the correct type. If so, executes the ticket's command; otherwise, passes it to the next handler.

If the handler can handle the object, the handler will be able to use the information of the ticket and pass it to execute the execute command linked to the ticket. When the handler handles the ticket, it will run the linked command.

Hint: look at the getType function to help make decisions

Note:

1. _ represents a space
2. <..> refers to some information that should be brought into the output
3. <newline> is a newline operator

The following Logic applies per handler:

* FinanceSupport:
  - Can only handle tickets with a FinanceCommand
  - If FinanceSupport can handle the ticket, before running the command, the handler will output:

```
Finance_Support_is_handling_the_ticket...<newLine>
```

  - If FinanceSupport cannot handle the ticket, print the following before passing on the ticket

```
Finance_Support_is_passing_on_the_ticket...<newLine>
```

* TechSupport:
  - Can only handle tickets with a TechCommand
  - If TechSupport can handle the ticket, before running the command, the handler will output:

```
Tech_Support_is_handling_the_ticket...<newLine>
```

  - If TechSupport cannot handle the ticket, print the following before passing on the ticket

```
Tech_Support_is_passing_on_the_ticket...<newLine>
```

* GeneralSupport:
  - Can only handle tickets with a GeneralCommand
  - If GeneralSupport can handle the ticket, before running the command, the handler will output:

```
General_Support_is_handling_the_ticket...<newLine>
```

  - If GeneralSupport cannot handle the ticket, print the following before passing on the ticket

```
General_Support_is_passing_on_the_ticket...<newLine>
```

- If none of the concrete handlers in the chain can handle the ticket or when there is no command linked to a ticket, ensure to output:

```
Ticket_Unhandled<newLine>
```

4.9 SystemMaker

This class defines the interface for constructing the ticket system.

- functions:
  - `reset(): void`
    Resets the builder
  - `addCategory(name: string): void`
    Adds a category.
  - `addFinanceTicket(id: string, info: string): void`
    Adds a finance ticket.
  - `addTechTicket(id: string, info: string): void`
    Adds a Tech ticket.
  - `addGeneralTicket(id: string, info: string): void`
    Adds a General ticket.

4.10 TicketSystemMaker

Implements the SystemMaker interface to actually build the TicketSystem.

- members:
  - `root: TicketSystem*`
    The root of the TicketSystem (composite) structure.
  - `currentCategory: TicketSystem*`
    The pointer to the category currently being built.
- functions:
  - `reset(): void`
    Resets the builder, by assigning the root member variable to a new category with the name "root" and assigning the current category member variable to the created category.
  - `addCategory(name: string): void`
    Adds a new category, using the name parameter, to the root category and assigns the current category member variable to this new category.
  - `addFinanceTicket(id: string, info: string): void`
    Create a TicketLeaf with the passed-in info and id, and set a newly created FinanceCommand to the cmd member variable of the TicketLeaf. The ticket must also be added to the current category.

- `addTechTicket(id: string, info: string): void`
  Create a TicketLeaf with the passed-in info and id, and set a newly created TechCommand to the cmd member variable. The ticket must also be added to the current category.
- `addGeneralTicket(id: string, info: string): void`
  Create a TicketLeaf with the passed-in info and id, and set a newly created GeneralCommand to the cmd member variable. The ticket must also be added to the current category.
- `getTicketSystem(): TicketSystem*`
  Returns the constructed ticket system. (Root)

4.11 Director

Uses the System Maker to create pre-defined ticket systems.

- members:
  - `maker: SystemMaker*`
    It is the system maker that will be used in the director.
  - `financeTickets: vector<TicketInformation*>`
    A vector containing objects that contain ticket information that will be used to create a finance ticket
  - `techTickets: vector<TicketInformation*>`
    A vector containing objects that contain ticket information that will be used to create a tech ticket
  - `generalTickets: vector<TicketInformation*>`
    A vector containing objects that contain ticket information that will be used to create a general ticket
- functions:
  - `Director(maker: SystemMaker*)`
    A constructor that sets the maker on initialization.
  - `getFinanceTickets(): vector<TicketInformation*>`
    Returns the vector of finance tickets. (This will follow the same logic for getTechTickets and getGeneralTickets)
  - `setFinanceTickets(financeTickets: vector<TicketInformation*>): void`
    Sets the financeTicket member variable to the passed in variable (This will follow the same logic for setTechTickets and setGeneralTickets)

– `construct(): TicketSystem*`
Constructs a ticket system using the saved maker, and the following steps will be taken into account when creating the ticket system:

1. Create a clean TicketSytem only containing the root
2. If there are Finance Tickets in the vector,
   (a) Add a category "Finance"
   (b) Add each ticket as a Finance ticket
3. If there are Tech Tickets in the vector,
   (a) Add a category "Tech"
   (b) Add each ticket as a Tech ticket
4. If there are General Tickets in the vector,
   (a) Add a category "General"
   (b) Add each ticket as a General ticket

5 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10 marks of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 2:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

60%-80%	80%
80%-100%	100%

Table 2: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

6 Implementation Details

- Do not include using `namespace std` in any of the files.
- You may only use the following libraries:
  - string
  - iostream
  - cmath
  - cstring
  - vector

7 Upload Checklist

The following C++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- `TicketInformation.cpp and .h`
- `TicketCommand.cpp and .h`
- `FinanceCommand.cpp and .h`
- `TechCommand.cpp and .h`
- `GeneralCommand.cpp and .h`
- `TicketSystem.cpp and .h`
- `TicketCategory.cpp and .h`
- `TicketLeaf.cpp and .h`

- `Support.cpp and .h`
- `FinanceSupport.cpp and .h`
- `TechSupport.cpp and .h`
- `GeneralSupport.cpp and .h`
- `SystemMaker.cpp and .h`
- `TicketSystemMaker.cpp and .h`
- `Director.cpp and .h`
- `main.cpp`
- Any textfiles used by your `main.cpp`

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

8 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ -std=C++11 -g *.cpp -o main
```

and run with the following command:

```
./main
```

Your final submission's mark will be your final mark. Upload your archive to the Practical 4 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**