|  |  |  |
| --- | --- | --- |
|  | Faculty of Computing, Engineering and Science |  |

**Assessment Cover Sheet and Feedback Form** 2018-19

|  |  |  |  |
| --- | --- | --- | --- |
| Module Code:  CS2S560 | Module Title:  Data Structures and Algorithms with Object Oriented Programming | | Module Team:  Emlyn Everitt, Janusz Kulon |
| Assessment Title and Tasks:  Poptart Dispenser | | | Assessment No.  2 |
| Date Set:  24-Sep-2018 09:00 | | Submission Date:  15-Mar-2019 23:59 | Return Date:  12-Apr-2019 23:59 |

**IT IS YOUR RESPONSIBILITY TO KEEP RECORDS OF ALL WORK SUBMITTED**

|  |
| --- |
| **Marking and Assessment** |
| This assignment will be marked out of 100%  This assignment contributes to 50% of the total module marks. |
| **Learning Outcomes to be assessed** (as specified in the validated module descriptor [https://icis.southwales.ac.uk/](https://icis.southwales.ac.uk/studentmodules/13599/studentmodulespecifications) ):  1) Demonstrate knowledge, comprehension and discernment in the efficient application of common data structures and algorithms, and collections.  2) Demonstrate knowledge, comprehension and discernment in the efficient application of object-oriented programming. |
| *Provisional mark only: subject to change and / or confirmation by the Assessment Board* |

## Source (100 marks)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Criteria** | | **Professional**  **(1st Class)** | **Senior**  **(Upper 2nd Class)** | **Intermediate**  **(Lower 2nd Class)** | **Amateur**  **(3rd Class)** | **Novice**  **(Narrow Fail)** | **Beginner**  **(Fail)** |
| **All Source Code (100 marks)** | **Design,**  **Structure & Efficiency** | Program is designed in a logical manner. Control structures are used effectively and correctly. Data structures, algorithms and patterns are implemented efficiently and appropriately. | Program is generally designed in logical manner. Control structures are used correctly. Data structures, algorithms and patterns are implemented effectively. | Program design is mostly logical. Control structures are used correctly, although more appropriate structures may have been selected. Reasonable data structures, algorithms and patterns are generally implemented. | Program isn’t as clear or logical as it should be. Control structures are frequently used incorrectly. Steps that are obviously inefficient are used. | Program isn’t as clear or logical as it should be. Control structures are occasionally used incorrectly. Steps that are clearly inefficient are used. | No meaningful evidence of logical, efficient or effective decision making in terms of selected data structures, algorithms and patterns. |
| **Comments** | Initial comments are complete. Internal documentation is complete and well suited to the program. Comments clarify meaning where needed. | Initial comments are nearly complete. Internal documentation is nearly complete and generally well suited to the program Comments generally clarify meaning where needed. | Initial comments are complete but internal documentation is in some small fashion inadequate. Comments usually clarify meaning. Unhelpful comments may exist. | Initial comments are incomplete or internal documentation is inadequate. Comments exist, but are frequently unhelpful or occasionally misleading. | Initial comments are incomplete and internal documentation is inadequate. Comments exist, but are frequently unhelpful or misleading. | Code is generally poorly commented with little regard to expectations. |

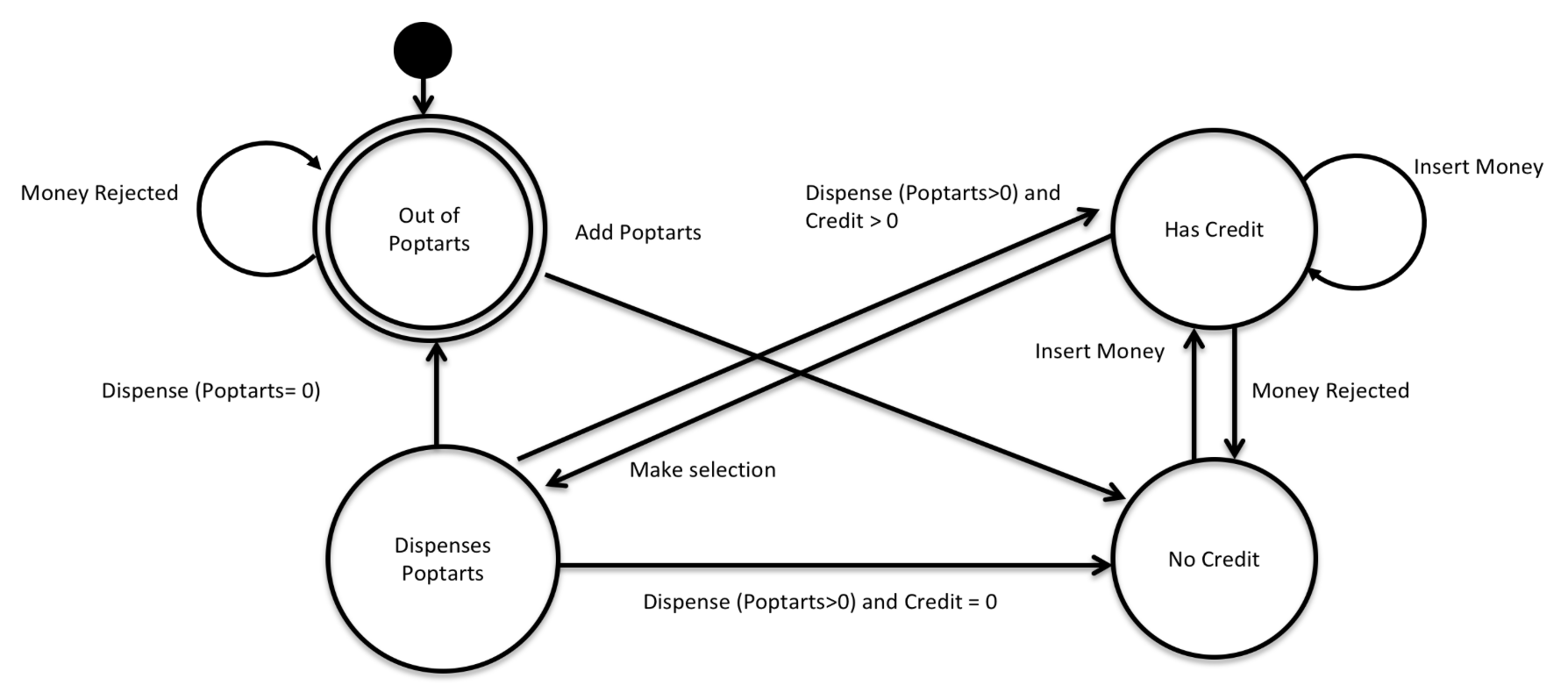
Tasks

## Scenario

You work for ChocDispeser LTD, a well-known manufacturer of chocolate dispensing machines. One day your boss comes to you to say that he wants you to develop the firmware for a ‘filled poptarts dispenser’.

Criteria for firmware:

* The firmware should only display basic success/ error messages: **no attempt should be made to implement user menus or prompt for user input in any way**.
* The firmware will only respond to method calls made to the public interface of Poptart\_Dispenser class. Note: methods will be expected to return a Boolean true when a requested operation results in a valid transition and a Boolean false when not – see appendix for further details)
* **None of the provided code (see appendix) should be broken in any way**. Implementation code can and will need to be added, but **the existing public APIs (methods) should remain the same** and **any provided code should continue to function as specified, unchanged**. **No changes should be made to specification of Public API under any circumstances**
* The firmware should implement the following state diagram without modification (i.e. you should not attempt to add or remove states / transitions).



* The state diagram and the API specification together represent the formal specification for the project, and a failure to adhere to them will result in project failure.
* You should employ appropriate design patterns in the delivery of this project. Any solution that is not implemented using appropriate design patterns will be judge a failure, even if it works.
* **DO NOT** **INCLUDE** **the ‘main’ function** in your submission. You code should be complete apart from this omission.
* The only libraries you should include are:<string> (for product description), <iostream> and <vector> (to support the storing of State classes within the StateContext class only – e.g. do not try to implement the Product/Filling classes using vectors). **No other libraries should be used.**
* **Throws should not be implemented**.
* **Your code must work.** If the code you present is broken in any way, marks will be lost.
* Because of the marks awarded / lost for functional/ non-functional code, **you would be better off submitting simple working code than more complex non-functional code.**
* When submitting your work, please submit your code in **a single .cpp file** to Unilearn in the format: *put\_your\_student\_ID\_here*.cpp – **DO NOT** submit the entire Visual Studio project direction and **DO NOT** submit a zip/rar file. Marks will be lost if you do.
* **Comments are** **IMPORTANT**. If code is not appropriately commented (see code comments section for further details) then many marks will be lost.
* The standard vending machine should dispense a range of different poptart bases in accordance with an option code passed to the Poptart\_Dispenser via the makeSelection method as specified by the following table:

|  |  |  |
| --- | --- | --- |
| Bases | Option Code | Cost |
| Plain | 1 | 100 |
| Spicy | 2 | 150 |
| Chocolate | 4 | 200 |
| Coconut | 8 | 200 |
| Fruity | 16 | 200 |

Multiple bases should not be supported: one base only.

## Bonus Section

Bonus marks will be awarded for the implementation of fillings. Your vending machine should allow the addition of one or more fillings to your poptart, with additional descriptions and costs being calculated and made available via the Product interface using **an appropriate design pattern.**

Option codes for fillings are as follows:

|  |  |  |
| --- | --- | --- |
| Fillings | Option Code | Cost |
| Chocolate | 32 | 20 |
| Banana | 64 | 50 |
| Strawberry | 128 | 50 |
| Raspberry | 256 | 50 |
| Apple | 512 | 50 |
| Blackberry | 1024 | 50 |
| Maple | 2048 | 100 |
| Marshmallow | 4096 | 20 |
| Cheese | 8192 | 70 |
| Cheese and ham | 16384 | 100 |
| Caramel | 32768 | 20 |
| Vanilla | 65536 | 50 |

**Please note: If you are unable to implement this section, ignore these additional option codes in your code and just return the base.**

## Mega Bonus Section

Additional bonus marks will be awarded for the implementation of the following, which will form part of the Product class public API (see appendix):

Product\* ReturnHighestCostItem(void){} //returns a pointer to the high cost item in a filled poptart (filling or base). Should return nullptr if not implemented

**Please note: If you are unable to implement this section, leave this code unchanged.**

## Code Comments

It is of particular importance that your code is well commented. The comments you make will form your documentation for this assignment and the quality of those comments will contribute significantly to your overall mark. Comments should attempt to describe the functionality of each section of code and how it fits into the overall behaviour of the program rather than just superficially describing what each line of code does in isolation.

e.g.

**Right**

x++; // x is the iterator used to keep track of the position in the data structure

//which is incremented each time the code loops

**Wrong**

x++ //x is incremented

It should also be noted that the marks awarded for comments will be weighted in favour of the feature complexity they are attempting to describe; with comments successfully describing the more advanced features resulting in the most marks being earned.

# Appendix A

#include <vector> //only to be used to contain dispenser states and nothing else

#include <iostream>

#include <string>

using namespace std;

enum state { Out\_Of\_Poptarts, No\_Credit, Has\_Credit, Dispenses\_Poptart };

enum stateParameter { No\_Of\_Poptarts, Credit, Cost\_Of\_Poptart };

class StateContext;

class State

{

protected:

StateContext\* CurrentContext;

public:

State(StateContext\* Context)

{

CurrentContext = Context;

}

virtual ~State(void)

{

}

};

class StateContext

{

protected:

State\* CurrentState=nullptr;

int stateIndex=0;

vector<State\*> availableStates;

vector<int> stateParameters;

public:

virtual ~StateContext()

{

for (int index = 0; index < this->availableStates.size(); index++) delete this->availableStates[index];

this->availableStates.clear();

this->stateParameters.clear();

}

virtual void setState(state newState)

{

this->CurrentState = availableStates[newState];

this->stateIndex = newState;

}

virtual int getStateIndex(void)

{

return this->stateIndex;

}

virtual void setStateParam(stateParameter SP, int value)

{

this->stateParameters[SP] = value;

}

virtual int getStateParam(stateParameter SP)

{

return this->stateParameters[SP];

}

};

class Transition

{

public:

virtual bool insertMoney(int) { cout << "Error!" << endl; return false; }

virtual bool makeSelection(int){ cout << "Error!" << endl; return false; }

virtual bool moneyRejected(void){ cout << "Error!" << endl; return false; }

virtual bool addPoptart(int){ cout << "Error!" << endl; return false; }

virtual bool dispense(void){ cout << "Error!" << endl; return false; }

};

class PoptartState : public State, public Transition

{

};

class OutOfPoptarts: public PoptartState

{

};

class NoCredit: public PoptartState

{

};

class HasCredit : public PoptartState

{

};

class DispensesPoptart: public PoptartState

{

};

class Product

{

protected:

string product\_description;

int itemCost = 0;

public:

virtual void consume(void)

{

}

virtual int cost(void){return this->itemCost;}

virtual string description(void){ return product\_description;}

virtual Product\* ReturnHighestCostItem(void)

{

}

};

class Base : public Product

{

};

class Filling : public Product

{

public:

virtual void fillProduct(Product\* NewBase);

virtual int cost(void);

virtual string description(void);

Product\* ReturnHighestCostItem(void){}

};

class Poptart\_Dispenser: public StateContext, public Transition

{

public:

Poptart\_Dispenser(int inventory\_count);

~Poptart\_Dispenser(void);

bool insertMoney(int money);

bool makeSelection(int option);

bool moneyRejected(void);

bool addPoptart(int number);

bool dispense(void);

Product\* getProduct(void);

virtual void setStateParam(stateParameter SP, int value);

virtual int getStateParam(stateParameter SP);

};