Programming Paradigms

CMP2092M - Assignment

Contents

14475946 - Liam Berridge

Task One

Characteristic	Description	Advantage
Statelessness	Where a function, process or system keeps no track of previous interactions and stores no variables between processes.	Reduces the memory footprint of a system as well as the overall complexity as processes can run independently of others.
Recursive	Where a function is able to call itself as many times as needed to resolve a process successfully.	Reduces the time complexity of an operation, easy implementation of nested loops.
Modularity	The capability of a process segmented, separated and recombined to achieve different results.	Ability to repurpose/reuse segments of code without creating redundancy, also allows testing of individual modules without requiring a complete program.
Immutability	The ability of a function to modify its own state or variables after initialisation.	Allows for changes in a function without creating new instances of said function, thereby having lower memory usage and being more efficient.
Polymorphism	The ability of a single entity or process to interface with multiple other entities or processes seamlessly.	Allows for greater expandability and efficiency of processes as they are able to work together without extra processes needing to be put in place to convert data.
Control Abstraction	The ability of a system to recognise some basic operations and be able to perform them without requiring additional programming.	Systems become easier to use as low level operations are already in place and can be somewhat neglected, allowing greater focus on the functionality of the program at large.
Encapsulation	The ability for a program to hide certain values, variables and functions which are not required for the module to be used.	By encapsulating certain processes and functions, they become easy to use but not critical to understand, and stop other modules or processes from modifying properties of the encapsulated module and damaging the program inadvertently.
Control Flows	The ability for a system to make decisions regarding branching where multiple variables may be passed.	Allows for automated systems to make complex systems efficiently instead of requiring the user to possess intricate knowledge of the system to use it.
Inheritance	The implementation of a hierarchy to replicate and distribute public and private elements of an object easily and efficiently when dealing with large amounts of entities or data.	Easily replicate complex systems and modify or extend them whilst keeping the original or other systems intact.
Procedural / Algorithmic	Where a process is required to perform several processes to function and produce a step by step solution	Allows for complex process chains to be achieved without recursion or large sections of code, separate processes or steps can be reused for other algorithms or processes too.
Object Composition	The ability of a system to recompose basic elements and data types into more complex ones.	Allows for a greater system which would be hard to compose of a single component, whilst also remaining modular meaning components can be swapped out if desired without breaking the whole system.

Referential Transparency	The ability for a function to operate without affecting the program outside of the operation/function	Gives a function or system reliability as it gives the same result regardless of the input.

Characteristic	OO	Imperative	Functional	Logical/Declarative
Statelessness	☒	☒	☑	☑
Recursive	☒	☑	☑	☒
Modularity	☑	☒	☒	☒
Immutability	☑	☒	☑	☒
Polymorphism	☑	☒	☒	☒
Control Abstraction	☑	☒	☑	☒
Encapsulation	☑	☒	☑	☒
Control Flows	☑	☑	☒	☒
Inheritance	☑	☒	☒	☒
Procedural / Algorithmic	☑	☑	☒	☑
Object Composition	☑	☒	☒	☒
Referential Transparency	☒	☒	☑	☑

Task Two

OO

Object-Oriented paradigms are aimed at creating objects or entities which are composed of various elements which can be manipulated to change the entity itself or its function. This allows for the creation of hierarchies of objects which may differ greatly but be replicated easily and quickly and may interact with other objects easily using interfaces.

Generally OO can be seen as a more appealing paradigm due to its incorporation of lots of different characteristics and its adaptability to a large number of different applications and scenarios, however this also creates issues for the same reason as having lots of characteristics. OO systems are complex and require large amounts of effort to create, they can also be slower when compared against simple logic or imperative systems as they are generally much larger programs overall; this also points out the issue of size as the programs require higher memory capacities to be able to run successfully.

Imperative

Imperative (Procedural) paradigms aim to create a solution using a state-changing method or routine and focuses largely on algorithmic or procedural processes, state changes and how the program itself operates under these changes.

Imperative languages can be very advantageous as they are generally very efficient, whilst also being familiar to use, similar to following a recipe or any other set of steps to achieve a goal. However despite this there are still disadvantages which follow on from that, such as a strict order of processes, meaning if the order is disrupted the entire system can break down or give out unexpected results which can be possibly dangerous if the results are not dissimilar to those expected. Referential transparency also does not hold in imperative programs, meaning the output can change despite having the same input; this can make understanding the underlying semantics harder once the system is in place.

Functional

Functional paradigms aim to create a formal solution to problems using mathematics, whilst avoiding state-changing instructions and mutable data. It has the basic form of Input, Process, and Output and can use recursion to repeat this order until a solution is reached. Due to this functional paradigms are dependant entirely on the input, and not the process itself.

Functional languages possess various features which can be advantageous to using such a paradigm when creating programs, examples of these features are control abstraction, where minor details are left to the system to deal with, meaning greater focus can then be placed on larger, more complex details; however lots of variables or instructions can be difficult to handle, and may be better off dealt with by using OO or Imperative languages.

Logical/Declarative

Logical paradigms are based upon logical premises, proof and rule-based systems which can replicate knowledge through use of expert systems. Generally these systems use logic sets to govern rules and laws in problem cases to reach a viable solution similar to how logical circuits work.

Whilst there are some benefits to using a logical or declarative approach to a problem, such as flexibility of the program or the ease of understanding, the major benefit of the paradigm is the system solving the problem itself efficiently, meaning the programmer only needs to understand the possible steps to a solution and how to configure them in the system.

Conclusion

In conclusion to the above, functional appears to be the most suited to the given problem, as it allows for recursion, mutability and referential transparency. These characteristics are important for the given system as the functions required by the user will utilise each of these and below is a justification of each of these characteristics to further justify the choice of functional.

Task Three

Name & Description

Text Editor – An ADT for a simple text editor with basic functionality.

Functions

Input character
Move cursor forward through text
Move cursor backwards through text
Move to start of sentence
Move to end of sentence
Move to start of previous word
Move to start of next word
Delete next character
Delete previous character
Highlight next character
Highlight previous character
Replace highlighted
Highlight word at cursor
Highlight next word
Highlight previous word
Highlight all
Copy Highlighted
Cut Highlighted
Paste Highlighted
Delete Highlighted
500 Character Limit (With error message)

Sets

T, the set of Text Editors {[Q],[E],[H],[X]}
Q, the set of Pre-Cursor Text ($\forall Q.Q \in RC$)
E, the set of After-Cursor Text ($\forall E.E \in RC$)
X, the set of Highlighted Text ($\forall X.X \in RC$)
C, the set of Characters {A…Z U a…z U 0…9}

Syntax

Create:	$\perp \rightarrow T$	Delete:	$T \rightarrow \perp$
Init:	$T \rightarrow T$	Input Character:	$T \times C \rightarrow T$
Move cursor forward:	$T \rightarrow T$	Move cursor backward:	$T \rightarrow T$
Move to start of sentence:	$T \rightarrow T$	Move to end of sentence:	$T \rightarrow T$
Move to start of next word:	$T \rightarrow T$	Move to start of previous word:	$T \rightarrow T$
Delete next character:	$T \rightarrow T$	Delete previous character:	$T \rightarrow T$
Highlight next character:	$T \rightarrow T$	Highlight previous character:	$T \rightarrow T$
Highlight to start of sentence:	$T \rightarrow T$	Highlight to end of sentence:	$T \rightarrow T$
Highlight to start of next word:	$T \rightarrow T$	Highlight to start of previous word:	$T \rightarrow T$
Highlight all:	$T \rightarrow T$	Copy Highlighted:	$T \rightarrow T$
Cut Highlighted:	$T \rightarrow T$	Paste:	$T \rightarrow T$
Delete Highlighted:	$T \rightarrow T$	Word Limit Reached:	$T \rightarrow T \cup M$

Semantics

pre-create()::true
post-create(r) :: r = {[],[],[],[]}

pre-destroy((q,e,h,x), c):: true
post-destroy((_,_,_,_); r):: r = ⊥

pre-init((q,e,h,x), c):: true
post-init((_,_,_,_); r)::r = (q,e,h,x)

pre-inputCharacters((q,e,h,x), c):: true
post-inputCharacters((q,e,_,_); r)::r = (q^^[c],e,_,_)

pre-moveCursorForwards(q,e,h,x):: true
post-moveCursorForwards((q,e,_,_); r)::r = (q^^[head e],tail e,_,_)

pre-moveCursorBackwards(q,e,h,x):: true
post-moveCursorBackwards((q,e,_,_); r)::r = (reverse(tail(reverse q)),(head(reverse q)).e,_,_)

pre-moveSentenceEnd(q,e,h,x):: true
post-moveSentenceEnd((q,e,_,_); r)::r = (q^^e,[ ],_,_)

pre-moveSentenceStart(q,e,h,x):: true
post-moveSentenceStart((q,e,_,_); r)::r = ([ ], q^^e,_,_)

pre-moveWordForwards(q,e,h,x):: true
post-moveWordForwards((q,e,_,_); r)::r =

pre-moveWordBackwards(q,e,h,x):: true
post-moveWordBackwards

pre-deleteNextCharacter(q,e,h,x):: true
post-deleteNextCharacter((q,e,_,__); r)::r = (q,tail(e),_,_)

pre-deletePreviousCharacter (q,e,h,x):: true
post-deletePreviousCharacter((q,e,h,_); r)::r = (reverse(tail(reverse q)),e,[ ],_)

pre-highlightNextCharacter(q,e,h,x):: true
post-highlightNextCharacter((q,e,h,_); r)::r = (q,e,(head e).h,_)

pre-highlightPreviousCharacter (q,e,h,x):: true
post-highlightPreviousCharacter((q,e,h,_); r)::r = (q,e,(reverse(head q)).h,_)

pre-highlightSentenceForwards(q,e,h,x):: true
post-highlightSentenceForwards((q,e,h,_); r)::r = (q,e,e,_)

pre-highlightSentenceBackwards(q,e,h,x):: true
post-highlightSentenceBackwards((q,e,h,_); r)::r = (q,e,q,_)

pre-highlightWordForwards(q,e,h,x):: true
post-highlightWordForwards

pre-highlightWordBackwards(q,e,h,x):: true
post-highlightWordBackwards

pre-highlightAll(q,e,h,x):: true
post-highlightAll((q,e,h,_); r)::r = (q,e,q^^e,_)

pre-copyHighlighted(q,e,h,x):: true
post-copyHighlighted((_,_,h,x); r)::r = (_,_,h,h)

pre-cutHighlighted(q,e,h,x):: true
post-cutHighlighted((_,_,h,x); r)::r = (_,_,[ ],h)

pre-paste(q,e,h,x):: true
post-paste((q,e,_,x); r)::r = (q^^x,e,[ ],x)

pre-deleteHighlighted(q,e,h,x):: true
post-deleteHighlighted((_,_,h,_); r)::r = (_,_,[ ],_)

Author

Liam Thomas Berridge (14475946@students.lincoln.ac.uk) Friday, 25 March 2016

References

Referential Transparency -
Recursion Time Complexity -
Nested Loops -