Design Patterns Essay

By Liam Knights

Programing design patterns are a generic solution to problems commonly seen throughout programing. Design patterns are not actually code themselves but more of a type of pseudo code or a directive for how to fix the common problem or avoid it all together. Using design patterns allows for a faster development process by providing tested archetypes for development. Using these patterns may expose a problem that would have arisen later on in the implementation of the program. And the reusing of design patterns helps with readability of different programs for the people familiar with the pattern.

Usually people will only use certain software design techniques for a certain problem depending on what suits them, those techniques are harder to apply a broader perspective unlike a design pattern that happens to be a general solution to rather than a specific one that requires multiple line of real code that is tied to a one imp particular problem to explain it rather than a few lines of pseudo code that can be used for multiple more general problems. It also means that a developer may communicate with to others using a more well-known name for the implementation of code, in addition design patterns can also be improved upon to allow for changes in the future without those changes being drastic such as a one minor detail change instead of a whole new set of different steps.

The singleton design pattern and its purpose are right in the name this pattern is to ensure that a class only has on instance that delivers a global access point to it, it also is to initialize on first use. The problem that its fixes is if the application needs one, and only one instance of an object as well as global access being a necessity. To make use of the singleton design pattern first make the instance object responsible for creation, initialization, access and enforcement. Declare the instance as private and static data. Provide a public member function that uses al initialization code and gives access to it. The access function gets called whenever a reference to the instance is required. Example of a singleton class

#include <iostream>

using namespace std;

class Number

{

public:

// 2. Define a public static accessor func

static Number \*instance();

static void setType() {};

virtual void setValue() {}

virtual int getValue(){}

protected:

int value;

// 4. Define all actors to be protected

Number() {};

// 1. Define a private static attribute

private:

static Number \*instance;

};

The iterator design pattern, the iterator design patterns purpose is to allow access to the elements of a combined object in sequence. The problem it fixes is the need to navigate different data structures so that like a list and the need for them to interface compatibly and you may want to go through the list a multitude of ways depending on the situation. An object like a list should utilize a way to access its various variables without exposing the rest of the its internal structure. Depending on the way you want to analyze this list may change for what you need to accomplish with it. That being said you do not want to extend the list interface with different traversals through the list, even if you could anticipate the operations you will need. That also being said you may need a to have more than on traversal for the same list so instead of contradicting yourself for different uses of the list a way of traversing or analyzing the objects that could change depending on what you need would be the way to go.

StackIter(const Stack \*s)

{

stk = s;

}

void first()

{

index = 0;

}

void next()

{

index++;

}

bool isDone()

{

return index == stk->sp + 1;

}

int currentItem()

{

return stk->items[index];

}

StackIter \*Stack::createIterator()const

{

return new StackIter(this);

}

The iterator pattern allows for change depending on what your need. The main idea is to place the traversal code outside of the list or list function and place it in an iterator function that sets a standard traversal of the list function and its various elements. The iterator function is fundamental to generic programing which is a strategy that aims to separate the idea of algorithm from the idea of data structure. The motivation behind this is to promote component based development and reduce configuration management. An example of the iterator design pattern

Prototype design pattern this patterns intent is to specify the kinds of functions to create using a prototype. It is meant to solve the problem of using the new keyword to much by creating each new function in a separate class all deriving from one big base class. To start declare a base class called something generic like entity make the functions virtual then create the other classes that are needed for the application and change the functions inside to change slightly depending on what your need similar to the iterator pattern but with multiple different variations of a function instead of one in particular function getting recalled for different events and doing all the work.

Class entity

{

Public:

Virtual void function()

{

//code goes here

}

};  
class user : public entity

{

Void function()

{

//slightly different code goes here

}

};

State design pattern, this is used to allow a function to alter its behavior when its internal state changes. The function will be seen as if it changed its class, it is an object-oriented state machine. A massive object must change its behavior depending on its state and this solves the problem of having to change the state constantly over the course of the application. In short, the state pattern is the solution to the problem of having to change behavior depending on its state.

The state design pattern is done by firstly, create two classes one abstract base class and another context class to present a single interface, represent the different states of the state each as a derived class of the state base class, define the specific state behavior in the correct classes, maintain a pointer to the current state in the context class. To change the state of the state machine to something else change the current state pointer to different state. The pattern does not tell where the transitions will be defined for each state. The choices are two the context class or each individual derived state class. The advantage of each induvial class is ease of adding new classes similar to it. The disadvantage is each class has knowledge of its siblings which can cause dependencies between classes.

Example of state pattern header files below.

#pragma once

#include "Renderer2D.h"

using namespace aie;

class StateMachine;

class State

{

public:

State();

~State();

virtual void OnEnter() = 0;

virtual void OnUpdate(float fDeltaTime, StateMachine\* pMachine) = 0;

virtual void OnDraw(Renderer2D\* m\_2dRenderer) = 0;

virtual void OnExit() = 0;

};

#pragma once

#include "Renderer2D.h"

#include "dyna.h"

#include "State.h"

#include "Stack.h"

using namespace aie;

class StateMachine

{

public:

StateMachine();

~StateMachine();

void Update(float deltaTime);

void Draw(Renderer2D\* m\_2dRenderer);

void PushState(int nStateIndex);

void AddState(int nStateIndex, State\* pState);

void PopState();

private:

dynamArray<State\*> m\_StateList;

Stack<State\*> m\_CurrentStack;

};

As useful as design patterns are, nothing is without criticism and multiple people in have criticized the use of design patterns. One such criticism is that it targets the wrong problems the saying things along the lines of the requirement of design patterns while using computer languages or techniques derives from the insufficient knowledge of the programmer and inability to come up with a solution on their own and that a solution to a problem should not be copied from another but thought up solely by the programmer. This however is pretentious, to think that if a solution exists and it solves a general problem coming up in programing from time to time then it shouldn’t be used, those solutions are how people identify problems with a language and how it evolves in to a later version of the same language.

Another criticism is that it lacks formal foundations this argument is probably derived from the fact that design patterns are not exactly code themselves and do not have a set structure to follow along to. This argument is the exact opposite of the previous in the fact that it wants them to show you the programmer exactly what to code and never think at all about how it actually uses these steps. This is similar arguing pseudo code should not have multiple keywords for each keyword in other languages but should have one and only one keyword and a strict set of rules to it, making it not pseudo code but real code.

Finally, to go over what has been stated already design patterns are not a strict set of rules but a general way of tackling problems that arise in programing. Singleton is used when you need one and only one instance of a function or object and tackles the problem of using it more than once. The iterator design pattern is to go through and traverse functions that act like a list by setting a standard function and choosing a way to traverse them without placing that code in the same function as the list function and stops the problem of recreating the code multiple times for different situations instead calls the iterator function. Prototype design pattern tackles the problem of multiple functions doing the same thing with slight changes by using a base class and virtual functions that if need be can be changed in the derived classes at the time needed, stopping the problem of using the same code twice in one application. The last pattern used is the state machine pattern stopping the problem of having to change the behavior depending on its state, instead changing its state over time depending on what you need. and before this some criticisms of using design patterns where stated like it targets the wrong problems and design patterns lack formal foundations.