**NODE DESIGN PATTERNS**

**CREATIONAL**

**Singleton**

Class Logger {

Log (message) {

Console.log(message);

}

}

Class Singleton {

Constructor() {

If (!Singleton.instance) {

Singleton.instance = new Logger();

}

}

getInstance() {

return Singleton.instance;

}

}

Const log = new Singleton().getInstance();

Const log2 = new Singleton().getInstance();

Console.log(log === log2); // true

**Prototype pattern**

* Similar to having a blueprint – similar object but with minor customisations

Class Shopper {

Constructor(name = “unnamed person”) {

This.name = name;

This.shoppingList = [];

}

setName(name) {

this.name = name;

}

getName() {

return this.name;

}

getShoppingList() {

return this.shoppingList;

}

addItemToList(item) {

this.shoppingList.push(item);

}

Clone() {

Var proto = Object.getPrototypeOf(this);

Var cloned = Object.create(proto);

Cloned.name = this.name;

Cloned.shoppingList = […this.shoppingList];

}

}

Var scout = new Shopper();

Scout.addItemToList(‘tent’);

Scout.addItemToList(‘map’);

Var ali = Scout.clone(); // ali now has the same shopping list thus far as *scout*

Ali.setName(‘Ali’);

Ali.addItemToList(‘Knife’);

**Factory**

* Define an interface for creating an object, but let subclasses decide which class to instantiate

Import Employee from ‘./Employee’;

Import Shopper from ‘./Shopper’;

Const UserFactory = (name, type) => {

Switch (type) {

Case ‘employee’:

Return new Employee(name);

Case ‘shopper’:

Return new Shopper();

}

}

Var ali = UserFactory(‘Ali’, ‘employee’);

Var lauren = UserFactory(‘Lauren’, ‘shopper’);

**Builder**

* Good for classes with multiple arguments in the constructor so you can see what is going on

Class Person {

Constructor (builder) {

This.name= builder.name;

This.age = builder.age;

This.gender = builder.gender;

}

isAdult() {

return this.age >= 18;

}

}

Class PersonBuilder {

Constructor(name) {

This.name = name;

}

Gender(gender) {

This.gender = gender;

Return this;

}

Age (age) {

This.age = age;

Return this;

}

Build() {

Return new Person(this);

}

}

Const ali = new PersonBuilder(‘Ali Issaee’).gender(‘male’).age(28).build();

// This looks better than if all was set in the contructor and instantiating like: new Person(‘Ali’, ‘male’, 28); especially if loads of integer values or Booleans which you couldn’t make out what they were doing, i.e. new Obj(true, false, 12.5, ‘happy’, false);

**STRUCTURAL**

**Adapter**

When you take an object, keep its interface but adapt it to a new environment or solution. Adapters let classes work together that couldn’t otherwise because of incompatible interfaces.

For example, the *localStorage* API is not available in node, an adapter would be able to create a *localStorage* class of its own, which does the same/similar actions to the *localStorage* client-side API, in order to work with node. It would need the same method names and properties.

**Proxy pattern**

A proxy is an object that controls access to another object. The intent of this pattern is to provide a placeholder for another object to control access to it.

A proxy must supply the same interface as the subject. The client would call the same methods that they would call on the actual object.

// Example below – file system proxy which only allows users to read markdown (MD) files.

Class FS\_Proxy {

Constructor(fs\_subject) {

This.fs = fs\_subject;

}

readFile(path, format, callback) {

if (!path.match(/.md$/i)) {

return callback(new Error(‘can only read markdown files’));

}

This.fs.readFile(path, format, error, contents) => {

If (error) {

Return callback(error);

}

Return callback(null, contents);

}

}

}

Const fsProxy = new FS\_Proxy(require(‘fs’)); // filesystem is an available node API

Const txtFile = path.join(\_\_dirname, ‘readme.txt’);

Const mdFile = path.join(\_\_dirname, ‘readme.md’);

Const result = (error, contents) => {

If (error) {

// code

}

Console.log(‘reading file…’);

}

fsProxy.readFile(textFile, ‘UTF-8’, result);

fsProxy.readFile(mdFile, ‘UTF-8’, result);

**Composite**

Compose objects into tree structures to represent part-whole hierarchies. Composites let clients treat individual objects and compositions of objects uniformly.

e.g. directory structure your folders would be branches and the files would be the leaves

class CatalogGroup {

constructor(name, composites = []) {

this.name = name;

this.composites = composites;

}

getTotal() {

return this.composites.reduce((total, nextItem) => {

total + nextItem;

}, 0);

}

Print() {

Console.log(this.name);

This.composites.forEach(item => item.print());

}

}

// where catalogItem has item name and price set in the constructor

Var boots = new CatalogItem(‘leather boots’, 79.99);

Var flipflops, var sneakers etc…

Var group\_shoes = new CatalogGroup(‘footwear’, [boots, flipflops, sneakers]);

Group\_shoes.print(); // the print method should also be in the catalogItem class

// multiple groups for other areas can also be created and then these groups can be put in to a bigger group of its own like full catalogue:

Var catalog = new CatalogGroup(‘full catalog’, [group\_shoes, clothing, keychains]);

Catalog.getTotal();

Catalog.print();

**Decorator**

Class InventoryItem {

Constructor(name, price) {

This.name = name;

This.price = price;

}

Print() {

Console.log(`${this.name} costs ${this.price}`);

}

}

Class GoldenInventoryItem {

Constructor(baseItem) {

This.name = `Golden ${baseItem.name}`;

This.price = 1.5 \* baseItem.price;

This.expensive = true;

}

//methods

}

A decorator does not need to have the same interface as the base class but it can.

Var Walkman = new InventoryItem(‘walkman’, 29.99);

Var goldenWalkman = new GoldenInventoryItem(Walkman);

**BEHAVIOURAL**

**Chain of responsibility**