Examförberedande

Oscar Palm

2021 hopefully

Contents

0.1 Explain object-oriented design principles . . . 3
0.1.1 Single-Responsibility Principle . . . 3
0.1.2 Open-Closed Principle . . . 3
0.1.3 Liskov Substitution Principle . . . 3
0.1.4 Separation of Concern . . . 3
0.1.5 High Cohesion, Low Coupling . . . 3
0.1.6 Interface Segregation Principle . . . 4
0.1.7 Dependency Inversion Principle . . . 4
0.1.8 Composition over Inheritance . . . 4
0.1.9 Law of Demeter . . . 4
0.1.10 Command Query Separation Principle . . . 4
0.2 Explain object-oriented design patterns . . . 5
0.2.1 What is a design pattern? . . . 5
0.2.2 Why do we want to use design patterns? . . . 5
0.2.3 Template Method . . . 6
0.2.4 Bridge . . . 6
0.2.5 Factory . . . 6
0.2.6 Chain of Responsibility . . . 7
0.2.7 Module . . . 7
0.2.8 Strategy . . . 8
0.2.9 Decorator . . . 8
0.2.10 Observer . . . 8
0.2.11 Iterator . . . 9
0.2.12 Facade . . . 9
0.2.13 State . . . 10
0.2.14 Adapter . . . 10
0.2.15 Model View Controller . . . 10
0.3 Explain basic object-oriented concepts . . . 12
0.3.1 Class and object . . . 12
0.3.2 Implementation inheritance . . . 12
0.3.3 Interface inheritance . . . 12
0.3.4 Variables and attributes . . . 12
0.3.5 Abstract classes and interfaces . . . 12
0.3.6 Static and dynamic types . . . 12
0.3.7 Dynamic binding . . . 13
0.3.8 Primitive types and reference types . . . 13
0.3.9 Overloading and overriding . . . 13
0.3.10 Encapsulation . . . 13
0.3.11 Aliasing . . . 13
0.4 Explain advanced mechanisms and techniques . . . 14
0.4.1 Polymorphism . . . 14
0.4.2 Sybtype polymorphism . . . 14
0.4.3 Parametric polymorphism . . . 14

0.4.4 Type constructors, type parameters and type arguments 14
0.4.5 Delegation 14
0.4.6 Subtypes and subclasses 14

Möjliga frågor och svar

0.1 Explain object-oriented design principles

0.1.1 Single-Responsibility Principle

"Just because you can doesn't mean you should".
A class should have one job only, letting the same class be responsible for too much can easily lead to unintended behaviour.

0.1.2 Open-Closed Principle

"Open-chest surgery isn't needed when putting on a coat".
When extending behaviour of a class/module/method it isn't recommended to directly modify the parent entity, instead create a subvariant of it, a child entity, and modify and/or extend the behaviour from within the child.
Not following this principle can easily lead to processes dependent on the parent involuntarily prematurely exitting.

0.1.3 Liskov Substitution Principle

"Derived types must be completely substitutable for their base types".
A class should only be made a subclass if it can be interpreted as an extension of the base class, some method modifications might be necessary but as soon as it starts to feel forced it's wrong; One should not partake in such behaviour as to force your code to do something they don't want to, if you start to think of your code as a living organism instead of an inanimate text file this should come naturally.

0.1.4 Separation of Concern

"do one thing and do it well. Good for general programming.
Consider one thing at a time and keep everyone else as abstract as possible, to better understand and decide how this one thing should be handled. "(PING)

0.1.5 High Cohesion, Low Coupling

What is cohesion

"The measure of the internal structure of a module. We want high cohesion, where all the components work together to solve the modle's responsibility. They should all be related to the same topic." (PING)

What is coupling

"How complicated the connection is between modules.
How likely a change in one module will cause a change in another module." (PING)

What it all means

"In order to have a good modular design, the modules should be as independent as possible. But they need a little coupling, due to the fact that we dont want them to not interact at all. If an object calls on another object's method, then those two are coupled. But the coupling needs to e as weak as possible. A strong dependency needs a good reason."(PING)

0.1.6 Interface Segregation Principle

"You want me to plug this in *where*!?".
No client should be forced to depend on methods it does not use.
All interfaces should be kept as small as possible, having too specific methods defined can lead to unnecessary code.
e.g. If you have an interface IPlayGuitar and in it specifies a method plugItIn(), that leads to acoustic guitars needing to implement a method only intended for electric guitars, this just causes unneccessary headaches. (As do writing this document)

0.1.7 Dependency Inversion Principle

"Would you solder a lamp directly to the electrical wiring in a wall?" - *Editors note: YES! that sounds awesome!*
Instead of having a high-level module depending on low-level modules (Having a module of a city depending on a specific implementation of the police station, having your hospital crash because of not the right kinds of doors) create an interface stating what the high-level module needs from the lower level one and have them implement them. If you follow this principle you needn't modify several high level modules when replacing one of the lower level ones, just make sure it implements the interface and simply hotswap them.

0.1.8 Composition over Inheritance

"Favor object composition over class inheritance"

"You use inheritance when it is an extension or specialization of the more abstract concept found in the superclass.
You use composition when there is a flow (data, business-logic) that you want to support within the confines of one class." (QUORA)

0.1.9 Law of Demeter

(Principle of least knowledge)
A method should only ever use objects it can access directly in its scope, it should never need to dig deeper through the layers of our security onion.

"A class should only talk to its close "friends"
A class should not be able to invoke methods from any but its close friends." (PING)

"We should never use objects inside of our parameters, instead have methods inside of the parameter objects which we can use to let the object itself modify or get data from its subobjects. Use only one dot!" (SAXEN)

0.1.10 Command Query Separation Principle

"A merhod should have either a side effect or a return value." (PING)

"A method should exclusively change the state of the program (side effects/command) or return a value (query).
All querys should be pure. It's okay to violate this in some cases: e.g. if we have a command and want to give feedback to the user about how the operation went." (SAXEN)

0.2 Explain object-oriented design patterns

0.2.1 What is a design pattern?

”Ett designpattern är en beprövad lösning på ett vanligt förekommande problem.” (HOM BRE)

”Design pattern - agreed upon solution to solve a problem. They are not code, they are design. Abstract templates for how a problem that keeps coming up can be solved. Principles are general qualities it should have, they are more abstract. Design patterns are more concrete.” (PING)

”A design pattern is a general solution to a common occurance within software design.
Can give us answers to questions such as when to use subclassing instead of interfaces or how we should structure and name our methods.

These answers might not always be the best ones but having patternized code at all leads to code that is easier to both read and modify.
” (SAXEN)

0.2.2 Why do we want to use design patterns?

”För att förbättra kod och strukturera, generalisera kod- och filstruktur” (FABBO)

0.2.3 Template Method

Describe the pattern

"We have different methods in different classes that are almost the same, but not quite. Create a superclass and put the code that is the same. Create an abstract method, and then call on the template method when implementing the abstract method." (PING)

In which situations can you use it?

Own answer: To break out the differing code from into a subclassdefinition of the superclass' abstract method in situations similar to the speedFactor()-method in the lab assignment we had, where the differing part is something which could be replaced with a non-trivial primitive value or where the differing part is a function call, like getNext() changed to getFirst() or getRandom().

Which components are included in it?

The necessary

What design problems can we solve with the help of it?

0.2.4 Bridge

Describe the pattern

"If we have two mostly identical classes, break out the differing functionality into own classes and compose the classses of these instead?"(SAXEN)

In which situations can you use it?

Own answer: If we are trying to define classes for a group of animals we would normally start with a superclass Animal, but what should it contain or not? Some animals can swim, some can speak, and some can fly; does this mean that animal should implement them? No! That would mean that fishes can fly and rabbits swim and we don't want that. We could break it down into different subclasses of Animal and have the animals themselves be subclasses of this instead, for example, have birds flying but not swimming etc., but then we have the odd cases of birds being able to swim and fishes eing able to fly, implementing it this way we then would need to implement a fly or swim method separately in these cases, and change them separately when we find the bugs that will exist in them. If we instead would limit the class Animal to only what all animals are capable of, like moving, procreating and dying, and then create and assign interfaces and compose the animals of classes holding information and methods on swimming or flying or eating, and then compose each animal of a set of classes and interfaces connected to these classes the code gets easier to maintain and it gets easier to check what an animal is capable of.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.5 Factory

Describe the pattern

"A factory is a (usually static) class whose methods abstract (hide) the behaviour of one or more concrete constructors.

- Reduces dependency on concrete class internal to packages. Now i can change the class, the factory object the creates objects of the new class.

- Often connected to an abstraction (ie superclass or interface)
- THen the same advantages as factory methods.

" (PING)

In which situations can you use it?

Own answer:In all cases where the program isn't dependent on a specific class but rather a variety of subclasses or classes implementing a specific interface, for example if we make a game and want to implement powerups spawning in our world, we more often than not don't care which specific powerup is supposed to spawn but rather want a random one, having a factory class handle which powerup will spawn and leaving the game controller unaware of which specific powerup spawned makes our code more comprehensive and abstract, removing a lot of complexity from the game controller.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.6 Chain of Responsibility

Describe the pattern

"a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain." (REFACTORING.GURU)

In which situations can you use it?

Own answer: One real world application of the Chain of Responsibility is modern day routing protocols where each router just forwards the package towards the router which pretends it's closest to the target, forwarding the problem until you reach the goal-neighbouring router which actually delivers the package.
More programmatically the pattern can be used when designing something like a graphical user interface, or GUI, in which a class handling user input firstly receives the request after which it's forwarded to some kind of controller which in turn forwards the request to a more task specific class, like a sound controller or player controller, and these in turn might handle the request themselves or forward them even deeper into the chain until a class feels that it's able to handle it correctly.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.7 Module

Describe the pattern

No information on refactoring.guru
Create coherent units on a higher level than e.g classes. Might not really be a design pattern.

In which situations can you use it?

Own answer: If you write a collection of classes which want to tightly control how they're used and in some way limit how you can access its contents, you can create a module.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.8 Strategy

Describe the pattern

"a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable." (REFACTORING.GURU)

In which situations can you use it?

Own answer: You have just completed your very own spell checking software and you feel amazing until you notice something; the program is only able to check for spelling errors in danish, but you want it to be able to check for errors in other languages as well, so you add functions for different languages as well. It turns out however, that this behemoth of a program, has gotten too large for a simple human to handle and this can't do but how are you going to be able to handle this problem?

The solution is to divide the different languages into own classes and wrap them together as subclasses under a spellcheck class umbrella.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.9 Decorator

Describe the pattern

"lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors." (REFACTORING.GURU)
Also called Wrapper pattern

In which situations can you use it?

Own answer: You have your very own blog software but since starting your blog you've noticed all your posts are riddled with spelling errors and upper and lower case letters in the wrong places, to fix this you create classes for spell correcting and to get the correct casing, but you know that since the blog program is quite complex you aren't going to be able to use these in every place where messages are posted,instead you choose to wrap the PostMessage-class in these classes, so as whenever you post a message it goes through the newly created classes and your text is corrected.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.10 Observer

Describe the pattern

"An interface called an observer. THe observer is something that can be connected to the observable. After the observer has subscribed to the model, the observer will be updated. It then displays the new status on the screen. This allows different views at the same time.

The observer is an interface, everything that implements the observer is an observer.

The signal object does not depend directly on signalview and signallogger.

When an object needs to notify other objects of events, without directly depending on them, broadcasts the events on an open channel, to which any interested object may register. "(PING)

”lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.” (REFACTORING.GURU)

In which situations can you use it?

Own answer: When creating the next version of MS paint you recognize that whenever you hold down left-click and move your mouse you want to draw on the screen but this requires multiple function calls in several different classes; the png needs to be updated, the screen needs to be updated, the brush needs to move and so on, you start out by writing each separate function call in the required methods but this is not just tedious, you notice several times that you forgot one or more function calls in some of the methods, therefore you decide to instead follow the observer pattern, instead of having explicit function calls everywhere you create a Observer class which most of your classes now are subclasses of, in this abstract class you define that all classes needs an update-method. Now whenever something needs to be updated you loop through your list of observers and let them decide on their own whether or not to do something.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.11 Iterator

Describe the pattern

”ets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).” (REFACTORING.GURU)
No

In which situations can you use it?

No

Which components are included in it?

What design problems can we solve with the help of it?

0.2.12 Facade

Describe the pattern

”Hide internal implementation. Make a package more abstract.

Libraries uses the facade patter. We have no idea how it is implemented, but it looks nice! Reduce dependency of client code and package.

Make it even more abstract by introducing an interface. We can even hide Polygon by creating an interface.

I can now switch polygon and its subclasses with anything! Adaption pattern! ”(PING)

In which situations can you use it?

Own answer: You have now constructed a really complex and hard to break encryption library, the only problem is it requires several hard to remember function calls and is in no way user friendly, therefore you decide to create a simplified interface by hiding EVERYTHING and creating one, single, public class which contains as few as possible methods which you can call and let them handle which internal components are needed, meaning instead of a dozen function calls the end user now only needs one to encrypt the data.
Facade pattern!

Which components are included in it?

What design problems can we solve with the help of it?

0.2.13 State

Describe the pattern

"State pattern, changing object after it has been created." (PING)
The created object is able to during runtime change how it acts depending on its state. Utilizes if- and switch/case-statements heavily.

In which situations can you use it?

Own answer: A website might need to show different elements depending on if the user is logged in, a guest or an admin. Depending on the user state the site might only be showed regularly, might contain user specific data, or even be allowed to be edited, this is a prime example of the state pattern.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.14 Adapter

Describe the pattern

"makes otherwise incompatible components compatible." (PING)

In which situations can you use it?

Own answer: When trying to use components created for slightly different purposes by different persons which looks like they should be compatible there might be some small detail making them incompatible, e.g. power sockets in different countries, instead of having to build one of the components from the ground up just to get a compatible version an adapter could be created and placed in-between them, like a power adapter or voltage converter.

Which components are included in it?

What design problems can we solve with the help of it?

0.2.15 Model View Controller

Describe the pattern

Very common, we want to seperate the data model (M) from the user interface (VC).
View - what the user sees
model - processsing
controller - input

In which situations can you use it?

Own answer: Whenever some kind of simulation needs to be interacted with in some kind of way you should use the MVC-pattern. In this case a simulation(model) can be anything from a backend server to a video game except the GUI.
The reasoning behind the MVC-pattern is that your model shouldn't crash just because your interface towards it does, imagine what would happen if Google crashed everytime someone googling managed to crash their browser.

Which components are included in it?

What design problems can we solve with the help of it?

0.3 Explain basic object-oriented concepts

0.3.1 Class and object

Class

A class is four main things;
Definition of a type, which later on instances of the class will be of.
Object containing static methods, constructors used to create instance objects and static variables.
Definition of what methods and attributes instances of the class are going to contain.
Description of what other classes and interfaces this type also is a type of.

Object

Class object: Only exists one of these per program, used to call static methods, access static variables and to create instances of the class.

Instance object: Created using a constructor contained in a class object. Contains everything defined in the class and its superclasses.

0.3.2 Implementation inheritance

A subclass of the superclass gets the superclass' implementation of its methods and attributes as long as they don't choose to override these.

0.3.3 Interface inheritance

The class gets a list of public methods it needs to implement for the program to be able to compile and run.

0.3.4 Variables and attributes

An attribute is a variable tied to a specific object.
A variable is a immutable value stored on a memory address defined by the program.

0.3.5 Abstract classes and interfaces

Abstract class

An abstract class acts just like a normal class with the exception that you can't access it directly or create objects of it, instead you create subclasses of it, where you need to implement its abstract methods or those too are abstract.

Interface

An interface is a code block where the programmer defines method signatures which needs to be used in every class which chooses to implement said interface.

0.3.6 Static and dynamic types

Static type

"A reference variable has a static type, the declared type (Polygon o = new Square(100,100). This will always remain the same." (PING)

Dynamic type

"the class of the object in the heap. This is the dynamic type and can change if the variable gets a new reference that points to a different object with a different type." (PING)

0.3.7 Dynamic binding

"the connection between a name of a method and the code is called binding. The binding is decided at runtime and it is decided by the type." (PING)

0.3.8 Primitive types and reference types

Primitive type

A primitive type is a value you can directly refer to or even statically use in the ALU.

Reference type

A reference type is a type where you only can use the data after taking a primitive type and using that to fetch data from the memory.

0.3.9 Overloading and overriding

Overloading

Overloading is the concept of giving several methods in a class the same name but different parameters, doing this we allow for a easier to grasp public and private interface to/in the class.

Overriding

When inheriting from a superclass we might encounter some methods which don't quite fit our subclass, we can then choose to implement our own version of it and override the superclass' version of it, calling our own instead.

0.3.10 Encapsulation

The practice of "hiding" methods, attributes ,or other programming related thingies from users.

0.3.11 Aliasing

Aliasing is the practise of pointing several reference pointers towards the same memory address, doing this might cause unexpected errors in the same way unsynchronized multithreading creates headaches.

0.4 Explain advanced mechanisms and techniques

0.4.1 Polymorphism

"An object can be used as if it was an object of the superclass." (PING)

0.4.2 Sybtype polymorphism

"Suppose S is a subtype of T. Anywhere a value of type T is required, a value of type S may be used."

0.4.3 Parametric polymorphism

"We can take in unspecified types A and B in the constructor. Then use variables of that unspecified static type. A and B are called type paramteres, Pair is a parametrized type. A and B can be used as concrete types. " (PING)

0.4.4 Type constructors, type parameters and type arguments

0.4.5 Delegation

The practice of letting other objects perform different tasks or calculations for you.

0.4.6 Subtypes and subclasses

Sybtype

A subtype of another type is, like with subclasses, a way to tell the program that this type contains everything that the superclass contains, but maybe even more; you should always be able to replace the type with one of its subtypes without it making any significant difference.

Subclass

An extension of another class.
Making a subclass of another class is the programming equivalent of printing a copy of a document and letting someone tippex and rewrite and/or add more pages to it.