**w05-01-DescribingFeatures-2024**

0:01  
OK, so this morning's lecture is on describing features.

0:08  
So as we said before, we're walking down one software iteration and we're going to stop and look at some more ideas around describing the system today.

0:18  
What's coming in future lectures is discussing implementation, unit testing, other types of testing, operation and maintenance.

0:27  
And then there's some other topics which we'll cover later around project management and quality.

0:33  
All right, So what have we done so far?

0:35  
We've been describing the system and with respect to this, we start off with the user interface design.

0:42  
We are trying to describe what the application will look like and this will hopefully be loosely coupled to our final software choice.

0:52  
So you want to design some interface that you think the customer wants, and then after you've done that, you think about the software.

1:00  
We also discussed the data model, and what we did last week in the lab anyway was that we started to describe the data model using the Entity Framework, sorry, an entity diagram rather, and we were using crow's foot.

1:15  
Now you can describe the data model using a UML class diagram or something else if you so wish.

1:22  
But the idea of this is that we're trying to describe what we think the data are that are flowing through the application.

1:29  
So not all of the data, but the crucial pieces, maybe the data that are stored in a relational database or the data that are displayed in the user interface.

1:40  
And we also discussed last week architecture and within this remit we were thinking about different architecture patterns.

1:47  
So for example, model, view controller, you will remember that one or the layered architecture.

1:52  
Anyway, there were lots of architecture choices which you can combine within an application to implement what you need.

2:01  
And now today we're describing functionality.

2:04  
So we're looking at other design tools that we can use to describe what we want to build.

2:11  
However, we're still trying not to say what actual technology we're using.

2:17  
And you can still do this without specifying the type of programming language, potentially depends on the nature of the programming language.

2:27  
OK, let's carry on.

2:30  
Now you may see in a piece of software design the concept of high level design.

2:36  
Now high level design is normally as described, very top level view what's going on and typically includes the architecture also includes any big services.

2:50  
So you might have a box somewhere that says the something service that does something, you know, it could be a data service, catalogue service.

2:59  
It's a big chunk of software that is replying to requests and providing data.

3:05  
And there may be large components within your software that have particular roles, which again would appear as rectangular boxes with some label telling the developers what it is they're meant to be doing with them.

3:18  
Now, low level design is around pretty much very much towards the implementation.

3:26  
Now you may be asked to design the API.

3:29  
Now here we're not thinking particularly about the user interface, but imagine you have several pieces of software in your bigger package and they all have to talk to each other.

3:43  
So these are internal APIs.

3:45  
These are the connection points between the different functions.

3:48  
Now, these are normally function calls, and you could have function calls in all sorts of different languages.

3:54  
It could be Python, it could be C#, it could be Java, Ruby, whatever.

3:59  
They're just function calls.

4:01  
So you can write these down.

4:02  
You can say what is going to be passed in to the application programming interface, what's going to be passed back.

4:09  
You can also describe what happens inside smaller pieces of code.

4:14  
Now this is potentially really useful when you have a complicated series of steps that have to take place.

4:20  
You can describe them in a diagram and then the developers all know what they need to implement.

4:26  
And then lastly, we can think about the actual classes.

4:29  
So we can describe if we want to use classes which we might not want to, we can describe what should be in the classes and then perhaps provide some implementation overview.

4:40  
But at this point, we're coming right up to actually implementing code.

4:45  
Now, whenever you're designing something, you need to have in mind time and money.

4:52  
It's always time and money.

4:54  
And if you spend too long designing something, then you are spending money where you didn't need to spend money.

5:02  
So there's a balance.

5:04  
You need to decide depending on the type of project, how much design you need.

5:10  
Now in a few few, sorry in previous lectures I've mentioned safety critical systems.

5:15  
So for example, if you're building software for civil nuclear, you will end up producing more design because a civil nuclear environment is obviously a dangerous environment.

5:26  
If you make a mistake, bad things could happen.

5:29  
So you end up producing more levels of design.

5:32  
You probably are going to be asked to produce a detailed low level design document, whereas other applications just won't need that or want that.

5:41  
Now the other thing you'll have noticed is that depending on the life cycle, the levels of documentation are different.

5:48  
So for example, if we use the waterfall or V life cycle, we end up with a large static collection of documents that are built up and then the software to match them is implemented.

6:00  
With Agile, we have smaller documents where we might go back and redesign things after discussion with the customer and then we update the software.

6:09  
However, in both cases we have to be careful that we don't spend too much time on the design because that's going to be costly.

6:20  
Now we always think about the high level design.

6:24  
Having no design at all is a very bad idea.

6:27  
You might in your head have a good idea of what you're going to do, but you then may, just on your own, without any other developers, end up creating a mess where you set off.

6:38  
It's a very complicated package.

6:40  
You build out function after function after function and then you realise, oh, that's not a very good structure, is it?

6:45  
No, it isn't.

6:46  
Oh dear, try to rewrite it.

6:48  
So that's just you, a single developer.

6:51  
If you have a set of developers, which is the normal, then they need the design to all know where they're going with the software.

7:00  
So we tend to limit however the detail of the low level design.

7:05  
When I have produced level design documents, it has been for either application programme interfaces that were going to be around for quite a long time and therefore they needed to be right or it was for safety critical systems.

7:20  
Those two setups is when I had to do it.

7:24  
Documentation, yeah, can be formed when you actually implement it.

7:29  
Remember we talked about potential of generating documentation from code.

7:35  
We will come back to this.

7:37  
It is a good idea to do.

7:38  
If it's easy, why not help somebody else?

7:43  
Yeah, and as I've already forewarned you, if you don't have any design, several developers, potentially you'll set off in different directions and then you end up with a real mess, which is more expensive.

7:55  
So it's a cost benefit balance.

7:58  
Now we're going to walk through a set of different diagrams and there are many diagrams, not all of which are in this talk, but I'm going to mention some of the more commonly used ones, how you might use them, and a few things about them.

8:12  
Now I'm assuming that after the lecture you'll go away and read about them and you might look at the textbook or you might look at other examples online.

8:20  
So it's not all in the lecture slides.

8:22  
Now the first one is the so-called UML use case diagram.

8:28  
Now this one is often used when you're trying to brainstorm user requirements, you're trying to document what it is that people will going to do with your software.

8:39  
So we can draw them and then show them to the client and we can say, is that correct?

8:44  
Is that all the actions that that thing or that person has to do?

8:48  
We can also use them when we're thinking about the user acceptance tests, we haven't come to those yet.

8:54  
They will be later on in the lecture series.

8:56  
And the user acceptance tests are there to verify does your software fulfil the implemented actions that you've set right.

9:05  
So you're ticking off, does it do this?

9:07  
Does it do that?

9:08  
Now it's much easier to create tests from the user's point of view.

9:13  
Now, what are they going to do with this software?

9:15  
Hence the use case diagrams are useful for that.

9:19  
All right, So what is in these use case diagrams?

9:22  
We have the use cases, so these, you'll see in a minute, are actions.

9:28  
So these are functions provided by the system that are needed by the user.

9:32  
So the user wants this thing.

9:36  
We then have actors.

9:38  
Yeah, funny nomenclature in the sense of the use of words, but this is just what they're called.

9:44  
Actors have a role.

9:46  
There are user, and rather like the user stories, they can actually be a persona.

9:51  
They don't actually have to be a real person, however they are normally a personal organisation that is a group of people, or they could be a persona.

9:59  
You can have subsystems where you have essentially a big use case and then inside you've got some other use cases, or you can group them up together.

10:11  
So this is large components within your application.

10:15  
And then you've got relationships where you draw lines to say this is related to that.

10:21  
So these are the symbols that are normally used.

10:23  
The actor on the left hand side here is a stick person.

10:28  
And below the actor you normally have the name of the actor, whether they're the user or the manager or whatever it is.

10:35  
And then the use case is normally in an ellipse as shown here.

10:40  
So you've got what it is you want to do in that ellipse and then you draw the relationships between the actor and the use cases with solid lines to say this actor needs this use case.

10:54  
And then the subsystem is a rectangle and you put all of the use cases inside this actors are outside of the subsystem.

11:05  
Now I have two other relationship arrows, 1 is include and the other one is extend.

11:09  
Include is slightly different to extend.

11:14  
So extend means that you are extending from another use case.

11:18  
However, the base use case is optional.

11:21  
Now this will become obvious in a minute on the next picture, but basically you you're going to have this use case and you might have the base one, whereas an inclusion is that you include the other use case.

11:34  
The base case is required.

11:36  
Now with all things pictures make it a lot easier.

11:39  
So this picture is from Wikipedia using the link at the bottom here.

11:43  
We have, it's not a software system, but it is an environment you may be familiar with that is of ordering food from a restaurant.

11:52  
So the actors here, we have a waiter, we have a client, we have a cashier, and we have a chef.

12:00  
So these are the main in actors in the environment.

12:03  
Now you could say, well hang on a minute, we have a washing up team and yeah, they're not in the diagram.

12:08  
Yeah, you're right, They're not in the diagram.

12:12  
So when you draw these diagrams, you might need to limit what you're drawing.

12:17  
Otherwise the diagram just becomes too busy.

12:19  
You might need several diagrams anyway.

12:22  
What can you see here?

12:23  
Well, you can see the waiter receives a food order.

12:27  
Notice we have a solid line where we also have the relationship between the actor and the use case written on the top.

12:37  
We've got the food order extends.

12:40  
A wine order, as I've already said, extend is it could include the base or not, meaning the person who's ordered the food could have ordered wine or they might not have.

12:54  
Here we can see again the serve.

12:57  
The food could include the serving of wine if wine was ordered.

13:02  
And there you can see a little condition written in the curly braces if wine was ordered here, the client eats the food and does drink the wine if that was ordered or if it was served rather.

13:14  
So it's working down from the top in this case.

13:16  
And then the client will pay for the food and pay for the wine if it was consumed, and then the cashier will accept the payment.

13:26  
That's it.

13:27  
So you can see from this, we've got actors, we've got relationships, we can add labels and all the text is relatively short so that it's easy to read, easy to visualise what's going on.

13:41  
All right, so that's a use case diagram.

13:44  
So we're now going to go on to some other types of diagram.

13:47  
The first one of this is a data flow diagram.

13:50  
Now, the idea of a data flow diagram is that we're trying to understand what data we have in our application and where are they going.

13:58  
This is very useful because often data flow that's in a mess makes the software hard to understand.

14:07  
I remember one piece of software that I, I was reviewing for a, a manufacturer of a particular machinery and the software designer had copied the configuration between classes and other structures.

14:23  
It was copied all the way around the place.

14:25  
And So what happened was if you changed it somewhere, you hadn't changed the original, it was just copied all the way around.

14:31  
And so it became very hard to understand what the state was of the application.

14:37  
So it's a good idea to think to yourself, where are the data coming from, what's going to happen to them and where they're going.

14:43  
So it's it's pretty simple.

14:45  
We have some data that's being received or sent, some process that's going to do something to the data.

14:53  
We have a data store where the data are coming from.

14:57  
And then we have a data flow where we're describing the flow between the data stores and the view potentially.

15:05  
So we can have this with or without object oriented design.

15:08  
We don't actually have to have object oriented design now.

15:12  
It's been around for quite a while this and it's sort of more or less in favour.

15:17  
There are lots of different standards of drawing the diagrams, so I've just used the Yordan code symbols.

15:24  
There are other symbols.

15:27  
So the idea here is that we've got a data store that's an external entity and we've got a process.

15:33  
And this time the lines are how the data are flowing.

15:38  
So here you can see we've got a catalogue and we've got some recommendations and they're flowing into this process request product, which then provides data to the shopper.

15:49  
Very simple, high level.

15:51  
Now you can break these down into many levels where you have what's inside of one of these processes.

15:57  
So you have hierarchy rather like Russian dolls.

16:01  
So the hierarchy tends to be set as the context diagram, meaning the top level.

16:08  
Then you can expand maybe one of these nodes, these processes to level 0 and then level 1.

16:15  
You don't normally go beyond level 1.

16:17  
Remember, design is a cost benefit balance.

16:23  
All right, so that's the data flow diagram.

16:25  
I'm going to mention the state diagram.

16:27  
Now, the data flow diagram is from thinking about how the data flow through your application.

16:32  
The state diagram is for describing the state of something.

16:37  
Now it's much more normal that this something could be a physical system, It might not be.

16:43  
If you think to yourself of a printer, the printer has several states.

16:48  
It could be idle, it could be an error, could be out of paper, it could be printing, it could be receiving data.

16:57  
So these are all States and it will move between these states depending on what happens to it.

17:04  
So what we have is we have a something, whatever it is, and then we have arrows that describe how we move between the states for that thing.

17:14  
Now we can, as we did with the previous diagram, have several levels of this where we look inside what's actually happening in a particular state change.

17:24  
Here are the little symbols.

17:27  
So we have a starting point, we have an ending point, we have a place where we can break, we can have forks and we can have joins.

17:34  
And this is the object, the thing we're changing the state of, and we can have a guard or a decision.

17:39  
You'll notice this looks very like a decision in a flow chart.

17:43  
It's the same sort of idea.

17:45  
And then the arrows take us from one state to another.

17:48  
So here's an obvious 1.

17:51  
So here we we're starting there and we're opening the door.

17:55  
Doors open.

17:56  
Great.

17:57  
We can close the door.

17:58  
Doors closed.

17:59  
We can lock the door, but then from locked, we have to go back to closed.

18:03  
We can't go locked and then open.

18:06  
We unlock the door, then it remains closed, then we open the door.

18:10  
Now we are going to have a bit of a go with this in the lab and play around with drawing a diagram just to make sure that we understand how to use it.

18:18  
But bear in mind this is a door.

18:20  
The door is not moving anywhere other than open closing and being locked.

18:25  
We've not introduced another object like there's not another person or something happening in here.

18:30  
It's all about one thing, the door.

18:35  
OK, so that's state diagrams, which are potentially quite useful for state, for bits of software that are managing systems.

18:42  
It could be a physical thing, or maybe not.

18:46  
Now we're going to discuss UML sequence diagrams.

18:49  
Now, these are useful for when you're just trying to describe what has to happen between layers of an application.

18:56  
Now, it could be that you're using this for a higher level design, or you could be using it for lower level design.

19:03  
I've used them mostly in lower level design, in fact, mostly for civil nuclear, where we had to describe what was calling what and how the data were returned.

19:15  
And again, it's a good process to cause you to think about what you are building.

19:22  
So for example, you may have a client and a server.

19:25  
You may have a web browser or something running on your laptop which is connecting to a server sending it data.

19:32  
You may have a layered architecture.

19:33  
You may have several API calls where you're calling different functions or services.

19:39  
Right.

19:39  
The UML syntax does actually allow you to have loops, so you can describe quite complicated things with UML sequence diagrams.

19:50  
However, we don't normally do this very much.

19:53  
We write a little bit loops just because it becomes more costly in terms of time.

19:58  
So you may use loops occasionally, but you perhaps just want to imply something and then actually just go ahead and implement it.

20:08  
Here it is.

20:09  
Very simply, we've got an object that is a thing.

20:14  
And what happens to The thing is it sends a message to another thing and then this comes back.

20:22  
So in this case we start from the dot and so we sent a message.

20:26  
So the user is typically here.

20:28  
So the user did something like it could say I want some data, maybe the list of students or something.

20:35  
This would dispatch the message to the web service, which would then RIP, send a message back.

20:41  
This is actually a call back.

20:43  
So in the case of a web service this wouldn't happen.

20:45  
A callback would happen potentially when you map to keyboard to a user interface and then you can return to the layer and then finally return.

20:56  
So the idea here is time flows from the top to the bottom.

20:59  
So you start here and then you are just following the arrows down.

21:04  
And whenever you've got one of these boxes, you're describing a session or a piece of memory that's being used by this particular layer.

21:13  
So you've sent some data here, and this layer keeps the data, whatever it's doing, in memory.

21:20  
And then when it's finished at this point, it's thrown it away again.

21:24  
So these rectangles here also describe scope a little bit like a function call where you have created some stuff, done some work, and you exit.

21:36  
Here's a more complicated 1.

21:38  
You need to limit the size of these diagrams because they can become quite messy.

21:43  
What you've got here are many layers.

21:45  
So you've dispatched, dispatched, dispatched.

21:48  
There's some self referencing layers, so on and so forth.

21:51  
Anyway, we're going to have another play with these in the lab, so hopefully you'll see a little bit more of them.

21:56  
But the the general idea is you're trying to help understand how we're going from different levels of the application and back again.

22:06  
Now these may be layers or they may actually just be function calls, depends on what kind of design that you're implementing if it's low level or higher level, right.

22:18  
Before I go on to CRC diagrams, does anybody have any questions or comments so far?

22:28  
No.

22:28  
OK, I'll carry on.

22:30  
I always find it useful to write down questions in case I forget them and then ask them later.

22:36  
So class responsibility collaborator diagrams, also called CRC diagrams.

22:42  
Now these are mentioned, sometimes they are used for object oriented design.

22:48  
If you don't need object oriented design, you don't need them now.

22:53  
They are a brainstorming technique where you're trying to quickly collect data and functionality into these CRC cards so that you can see what needs to be a class or not.

23:08  
And you may use post it notes or something like post it notes.

23:12  
The reference to them is at the bottom here.

23:16  
So what you do is you write the class name at the top.

23:19  
So what is it meant to be doing?

23:21  
You say what it what it's responsibilities are and you say how it interacts with other classes.

23:28  
So for example, the class name at the top could be something like, let's say it's authenticator.

23:35  
The responsibility would be to receive and process the password and the collaborator could be some information service somewhere else.

23:47  
So here is a an example.

23:49  
We've got a customer and this is a class.

23:53  
Remember, it's not a person.

23:55  
And so the responsibilities of the class are to place orders.

23:59  
They know their own names.

24:00  
So that means they have a name as a data member, has an address as a data member, has customer number and order history.

24:09  
And so it's going to collaborate with this other class called an order.

24:13  
So from this you could have a A use case potentially.

24:19  
You could also have a UML class diagram where you have described the order and the customer.

24:28  
So this is a brainstorming technique.

24:30  
If you can't go straight to UML, the class diagrams, you can try this and then draw the UML class diagram afterwards.

24:38  
OK, so we're jumping around between different diagram techniques and I'm trying to tell you when you might want to use them.

24:46  
We're now going to go for something quite low level, meaning very close to how the computer actually works, which is a control flow graph.

24:54  
Now this is particularly useful if you're working with compilers.

24:57  
So remember the compiler for Java takes the Java code and it compiles it down to bytecode, or if it's C, it compiles it down to assembler machine code.

25:07  
So what we do here is we've got logic and decisions, and we can have loops and we have arrows between these things.

25:16  
So this is quite old, 1970.

25:19  
So here is an example.

25:21  
On the left we have a while loop.

25:23  
This is implemented on the left in Python just because you know the language.

25:28  
So here we've got a while loop, and you can see the steps between the starting point.

25:33  
Then we go into the loop and the loop condition is here at the top, and this rectangle we go round and round, adding up #1 until we break out of the loop into our next step, which is to add 2 here.

25:47  
So you can see between the control flow graph and the actual implementation.

25:51  
It's very, very close.

25:53  
Now, you might want to do this if you're trying to puzzle out how a compiler is working.

25:58  
So it's a very low level tool.

26:00  
And I did actually have a student, not an MSC student, but there was an undergraduate working with me on decompiling, and she ended up using control flow graphs, which was the established way of doing this from a paper.

26:16  
Great.

26:17  
So we've gone super low level.

26:18  
Now we're coming back to the C4 model.

26:21  
Now the C4 model is an all-encompassing way of thinking about the architecture.

26:28  
So we have many architecture viewpoints in here.

26:32  
We have a so-called context diagram where we've got the system users and other users, container diagrams where we have interrelated containers, we have component diagrams where we've got these containers related, and then finally we have code diagrams.

26:50  
So the idea with the C4 model is they put together a hierarchy of design tools.

26:56  
So this is a very high level.

26:57  
Remember the context is the top level of the design and then the code diagram.

27:04  
This is really close to the implementation.

27:07  
You can read about it a bit more if you follow the link at the bottom.

27:11  
OK, here is an example.

27:13  
We have a customer and they are a person.

27:17  
So notice we have in the square brackets the type of the thing and at the top the name of the thing.

27:25  
So here is a customer and what are they doing?

27:28  
They are sending out a payment request and in here in this dotted box we have the bank.

27:37  
So the bank is an enterprise, it's a company.

27:41  
So the customer sits outside the enterprise and sends a message in to Internet banking software system.

27:47  
Notice we've got 123 software systems.

27:51  
This one sends a message to the banking system, which is probably some nasty low level cobalt system that you know, the bank invented in, I don't know, 1960 or whatever it was.

28:05  
And this is actually what makes the payment.

28:07  
And then it also sends a message to the e-mail system, which sends a confirmation e-mail back to the user.

28:16  
Now we can go from this top level, which is the context down to the container.

28:21  
And So what we've done here is we've greyed out the the things we had in the last picture.

28:27  
So here you can see we've got the banking system, the e-mail system, and they're both over here.

28:32  
We greyed out the customer and we're now breaking up this one.

28:39  
OK, we're breaking up the online piece of it.

28:44  
So here we have the customer again, and what are they interacting with?

28:47  
Well, they have to interact with some web application, but before they're allowed to do that, they've got a single web page application, yes.

28:57  
So they're going to log into the web application that delivers a single page application.

29:01  
Single page application would run in their browser.

29:04  
It would be JavaScript probably, maybe written in React or Flutter or something.

29:08  
And then the customer sends a message to the JavaScript running in their browser that then sends an HTTP method to another web service.

29:18  
So this web service here is providing the authentication and it's providing the static content.

29:24  
This one is providing the Jason, potentially Jason receive and return the messages.

29:32  
So it receives a message, it interacts with its own database, which here I've said could be a SQL Server.

29:40  
Fine.

29:41  
And then from here we have a message to the e-mail system and the banking system.

29:46  
So you can see that we can go from the top level to a step inside.

29:52  
OK, fine.

29:54  
We'll go on to design patterns.

29:57  
Before we think about design patterns, does anybody have any questions about any of the design diagrams so far?

30:05  
Yes, yes, a good, good question.

30:24  
I think when this, so when this is false, it will exit like that.

30:33  
When it's true, it just goes around and around.

30:35  
So here is it true?

30:36  
It's going to do this, then it's going to do that, and then this and then that and then this.

30:40  
So what's happening is we come in here, we go down there, this is finished, we come back.

30:46  
So this is essentially we're going around until you break out there and you're going to break out here when that isn't true.

30:54  
Yeah.

30:55  
So it's very minimalistic, but yeah, anybody else?

31:03  
OK, we'll go on now.

31:06  
Design patterns are actually starting to talk about how you might imlement things O it overlaps with what we're going to discuss next week.

31:18  
They are useful because you are trying to describe functionality in software, and potentially you can do it in a more efficient way where if you remember my example earlier, having copied the data all the way around your application and therefore not knowing which is the actual current data, you can avoid that sort of nonsense by having a useful approach to design patterns now.

31:42  
Design patterns were originally discussed or at least summarised in this publication I mentioned at the bottom in 1994, and they were mostly around object oriented design.

31:53  
Now there are some that are generally applicable or can be applied to other programming languages, and others are less so.

32:00  
So I'm just discussing some of the more useful design patterns that you may well see or want to use.

32:06  
If you want the rest, go and read the book.

32:09  
OK, so common design patterns.

32:12  
We often see decorators now.

32:15  
These tend to be used for web web applications.

32:20  
And for example, if you write a web application using Python Flask, or if you write 1using.net Web API, you'll see decorators and I'll show you an example in a minute, don't worry.

32:34  
But the idea is there's another way of finding the function.

32:38  
You're not directly calling it, but you are able to load the decorator dynamically and find the function and call it without actually directly linking to the function.

32:51  
So this is quite useful when you're connecting say a URL to a function call, which is the example I'm giving you.

32:59  
We have factory methods.

33:00  
Now a factory method is rather like it suggests it's job is to make objects.

33:06  
So you call it and it gives you another OB gives you another thing.

33:09  
You call, it gives you another thing.

33:11  
So these could be embedded in a service where you call the service and it keeps giving you things, or it could just be a function and it returns a new thing.

33:22  
We have iterators.

33:23  
Now you have seen these in Python, but you probably didn't realise they're iterators.

33:29  
An iterator allows you to jump between objects that are in a list, or it could be any kind of sequential container.

33:37  
So you're given the iterator and you say, can I have the next one or at least 4, whatever it is, and it will then jump to the next one.

33:46  
Now you don't have to worry about, well, it's inside the container.

33:50  
You don't have to worry about the complexity of how it's jumping around the memory.

33:54  
You're just given the iterator and you use it to jump between the objects.

34:00  
Singletons are really useful.

34:02  
The principle of a Singleton is that you can only have one of them in your programme at one time.

34:08  
You can't make another one.

34:09  
Now that's really handy because you can create that process and then when you want to say, access the configuration data, it could be in the Singleton.

34:21  
And oh, what, what have we got in the config file?

34:23  
Let's ask the Singleton.

34:24  
Oh great, tells us the answer.

34:26  
And when you're somewhere else over in the programme, you say what's in the config file, you ask the Singleton again.

34:32  
You're not going to create two or three classes just because you can't.

34:35  
The Singleton pattern prevents you from doing so.

34:38  
We'll see how to do it in the lab.

34:40  
By the way, with Java we can have template methods, and these appear in lots of languages.

34:46  
You see templates a lot in C++.

34:50  
They're quite useful.

34:51  
And the idea here is that we can implement the template with more than one type.

34:56  
It could be that we could have a float or an integer, but the functionality that we're going to use on it is the same.

35:04  
However, the type inside is different, so we template it and then we can specify the type.

35:10  
So for example, we could have a vector of floats or vector of integers, or we could have a vector of some kind of objects.

35:17  
As long as we can add them together or multiply them, it's OK, right?

35:23  
As I already said, decorators are often used with NET MVC.

35:28  
You'll see them used in NET Web API as well.

35:32  
You'll see them used with Python Flask.

35:34  
These are web application frameworks and the idea here is to map the URL to the function core.

35:42  
Great.

35:43  
Here's what it looks like with Python Flask.

35:45  
I'm just giving you something very simple so you can see how it works.

35:49  
What we have is we have a URL.

35:51  
That whole URL could be my lovely bookshop.com or whatever you like.

35:56  
And then after the slash, we've got a suffix, in this case books.

36:02  
And so we're mapping the word, the word books to an action.

36:06  
So if a user says my lovely bookshop.com/books, then that request will come to this a decorator, which is the books blueprint.

36:18  
And you can see here we're allowed to get books or post them.

36:22  
So that maps the incoming URL suffix to this function.

36:26  
The it's called a routing.

36:30  
Now, in Microsoft, this similar idea is perhaps a little less transparent, but it's the same idea with the.net frameworks.

36:39  
And then inside we've got some functionality which is just getting the books, turning them into Jason and returning them.

36:45  
But the point I'm trying to illustrate here is this is a decorator.

36:50  
Now, the way the word decorator is used between different programming languages isn't always the same.

36:56  
So decorators with Python don't completely work in the same way as the original definition, but they do, more or less, and they are actually known as decorators, which is why I'm calling them such.

37:07  
They factory, so this one is useful for services.

37:13  
Imagine you have some kind of database service, or it could be a web service and you are asking for a new thread, a new connection thread so you can get some data.

37:26  
We then hand out those threads and maybe we have to delete them at some point so we don't run out of memory.

37:33  
So here is a factory, and again, this is using a web service just because I'm already talking about one.

37:40  
This function creates a Flask instance.

37:44  
So why do you use this?

37:46  
Well, you use it because when you're using Flask web services, you normally use them with Unicorn or Green Unicorn as it's called, and it creates several threads of your Flask web service.

38:00  
You've got several parallel versions and so to give it a way to create the threads, you can give it a factory method.

38:07  
So this function, what happens inside the function, you can see it creates a Flask object there.

38:14  
Now it does some other stuff configuring it, connecting to the database and then at the bottom it returns the object.

38:21  
So every time you call this function, it's making and configuring a new object a returning.

38:26  
So it's a factory function, a Singleton.

38:32  
Now a Singleton is great for controlling data because the Singleton can only be made once.

38:39  
So it can have not necessarily just inside it, but it could have classes and other things inside it that contain the data.

38:46  
And whenever you need those data, you go to the Singleton because you can only have one instance.

38:51  
Now the way you implement this properly, as in the way it should be implemented, is with a private constructor, meaning nobody else can actually create the object.

39:01  
And then what you do is you have a static variable which whenever the static variable is retrieved it, when it's first retrieved, will create the object, and then after that just returns the static instance of the object.

39:16  
So you can't create another one, it's impossible.

39:20  
Now, it is possible to use singletons in many programming languages.

39:24  
So for example I've used them in C++ quite a bit.

39:28  
Yes, you can implement them in Python.

39:31  
Well, not really, It's with a module.

39:33  
So you can implement them properly in Java, which we'll see in the lab.

39:37  
You can implement them in C# I believe.

39:40  
Anyway, most object oriented languages will allow you to implement them.

39:43  
With Python.

39:45  
You can sort of implement a counted Singleton, but it's with a module, not with a class.

39:50  
And yeah, you have to essentially obey the same rules, which is that you have a static reference.

39:57  
Python doesn't allow you to protect yourself as thoroughly as Java or C# or C++ word in that you can kind of override the Singleton if you're not disciplined.

40:07  
Anyway, if you want to learn about the Java one, follow the link at the bottom.

40:14  
All right.

40:14  
We've rocketed through this lecture and so I've got plenty of time for questions and comments.

40:21  
We want to think about our design based on what we're building, and there must be a balance between the cost and the benefit.

40:30  
We can't have lots of time on the design if we didn't really need it.

40:35  
So what you may end up doing is create a user interface.

40:38  
Definitely a data design, yes.

40:41  
You may then end up implementing a first version, trying to prototype it, understand it.

40:46  
You may have a bit of low level design depending on what it is you're building.

40:50  
If it is a more complicated system, if it's going to implement with sorry interface with somebody's hardware, you may want to go for a little bit more design, particularly around how the components are working.

41:02  
State diagrams are really useful for actions, and physical things like this will be open or closed given some state.

41:10  
Something like that.

41:11  
OK, so we need to choose lastly the design patterns for the system.

41:17  
Don't go ahead and just Chuck all your code together and hope for the best.

41:21  
If you can use a design pattern that's going to make your code a bit more robust, do so.

41:27  
So for example, a factory method, it's sensible, makes sense, yes.

41:31  
OK, In this case, it's needed by Gunicorn, but it is a good idea to have a way of creating objects.

41:39  
So I'm going to stop there and ask if there are any comments or questions.

41:48  
Everybody's sleepy.

41:49  
Yeah.

41:57  
All right.

41:57  
Well, we've really got nothing.

42:00  
So we will try some of these in the lab, and then hopefully that will cause you to see how you use them of these diagrams in commercial contexts.

42:11  
I've had to use most of them.

42:15  
So to give you an idea, so sometimes people you ask how often do you use these use case diagrams are quite common.

42:25  
They're not always needed for requirements capture, but they're quite common and they may be not very complicated.

42:32  
You know, similar one page data flow diagrams.

42:38  
I have seen them used that was more civil nuclear thing.

42:42  
So, yeah, not that used.

42:46  
What else do we have?

42:47  
State diagrams.

42:48  
These, these are useful.

42:50  
Again, if it's safety critical, if it's physical systems, I've seen them used there.

42:55  
Yeah, complicated state we we'd end up using a state diagram.

43:00  
I have used these UML sequence diagrams quite a lot for higher level and lower level design.

43:07  
I for several different pieces of software.

43:09  
So it's a good idea to know how to use these.

43:11  
They're potentially quite useful.

43:13  
As I've already said, I've only used the loop a few times because you want to keep the diagram simple enough.

43:19  
It just fits on a page and somebody can read it.

43:21  
So typically speaking, I would cut the diagram, let's say 5 layers, and then maybe break out one of these layers into a different diagram if needed.

43:32  
So the very most complicated job I did with these kind of diagrams, we actually had several diagrams rather than just one with everything in it.

43:43  
CRC cards, I've never used them.

43:46  
Well, my former colleague used them quite a lot.

43:50  
People find it useful.

43:52  
A lot of these things are about what is useful to obtain the goal.

43:57  
So if you find CRC cards helpful for brainstorming classes, do it.

44:02  
If you don't, don't bother.

44:04  
Firstly, I don't, so I don't use them.

44:08  
Control flow graphs.

44:10  
These were really useful for when we were working on decompiling code.

44:14  
Other than that, I never used them.

44:17  
They're only really low level.

44:19  
The C4 model.

44:21  
I have done something a little bit like this.

44:23  
There's another concept which is perhaps a bit more advanced, which is where we have several architecture views and we start thinking about the logical and physical architecture.

44:34  
So I haven't really described that.

44:36  
I've sort of hinted at things that could be logical and physical.

44:41  
So yeah, I've touched on this a little bit, and that was for, yeah, a little bit more sensitive application.

44:48  
And then in terms of design patterns, what have I used?

44:52  
Decorators I end up using all the time with web services.

44:56  
Factory methods I do use quite a bit, probably not as much.

45:00  
Decorators.

45:01  
Iterators all over the place all the time.

45:03  
You're using iterators, you don't even know it.

45:06  
Singletons, yes, I have used these, as I already said, normally when it's controlling data flow and that obviously is going back to thinking about the data flow.

45:16  
So if you feel you need to draw data flow diagram, do so and then you can make sure your data are in the right place, you know what's happening to them.

45:25  
I have used template methods occasionally.

45:28  
They're not that useful.

45:30  
There is a kind of trap here where you can think to yourself, oh, I can do that with a template method guy.

45:38  
That would be really cool.

45:40  
Let's do it.

45:41  
And then somebody else comes along and says, what is this?

45:45  
Yeah.

45:45  
So you want to use a diagram, or at least you want to use a design pattern when it is helpful.

45:53  
So if you think about my vector analogy, having some sort of vector thing which will allow you to have different types that can work with it, that's a sensible template.

46:03  
And in fact it's part of the standard template library in C so it's a good idea.

46:09  
In other cases, you may not need a template method.

46:11  
So just think, do I really need one or not?

46:15  
And it probably you don't, they're not that common.

46:19  
Yeah, these are the implementations.

46:21  
OK fine, alright.

46:23  
If there's nothing else I will remain here for shy people.

46:27  
Other than that, I hope you have a good day.

46:29  
I'll look forward to seeing you later in the week in the lab.