Lecture Video Transcript

# CSE 4321 Security Testing (Part 1, Fall 2020)

00:02

the topic for this video is security testing first we have an introduction what

00:10

is security testing and what is unique about security testing next we look at a

00:20

couple of security vulnerabilities that we find a lot in practical applications then

00:30

we discuss fuzz testing which is a widely used approach for security testing software

00:44

security is about how to build software that is secure that is able to stand

00:55

against malicious attacks and there are different types of security properties including

01:06

confidentiality it is about keeping sensitive information private

01:13

integrity it is about making sure data is not

01:20

modified by people who are not allowed to modify

01:26

availability it is about making sure data and

01:30

services are available to the users

01:34

authenticity making sure the identity of a user

01:42

is authentic

01:44

authorization making sure resources can only be

01:49

accessed by people who are allowed to access

01:54

and other types of properties very important security must be built in

02:04

it means that we must take security into consideration in the beginning of a

02:13

project security is not something that you just add in the end of a

02:24

project the difference between functionality and security functionality

02:32

is about what software should do in contrast security is about what software

02:42

should not do security is typically considered to be a secondary

02:49

concern the reason is that no matter how secure a system is if

02:56

it does not do what it is supposed to do then the system is not useful which

03:08

one is more challenging typically for functionality we have a pretty good idea

03:17

about what a software system is supposed to do and in many cases we

03:25

have the specification document we have the requirement document we can look up

03:36

but in general we do not have a good idea about what software is not

03:43

supposed to do we typically do not have a document that specifies what the

03:51

system is not supposed to do so in this respect security could be more challenging

04:03

many security requirements are not explicitly specified for example we

04:10

have implicit requirements that any software system should not have buffer

04:18

overflow vulnerabilities should not have cross-site scripting vulnerabilities and

04:27

other types of vulnerabilities security attacks are

04:44

often possible because we do not do

04:48

adequate input validation in our program the reason is that the way an attacker

04:55

typically attacks a system is to try different invalid inputs inputs that are

05:06

not expected by the developer to compromise the security of a system so

05:16

in principle all the inputs especially inputs that are coming from the Internet

05:24

should be considered dangerous and needs to be checked for

05:29

validity before we process the input and data structures should be considered to

05:41

be tainted if the data structures use any information from the external inputs

06:02

security testing is about how to detect security vulnerabilities that may exist

06:10

in a program before the hackers do so when we do security testing we need to think

06:19

like a hacker we need to act like a hacker but we have a

06:27

good

06:28

intent we want to make the system secure we're not trying to break into

06:36

the system the challenge of security testing is how to automate the hacking

06:45

process that is to a large extent creative and we know many hackers break

06:55

into a system by coming up with a creative way to use the system by

07:02

coming up with some creative inputs they could give to the system

07:14

in general if you want to automate a creative process it can be very

07:21

difficult if the process is mechanical then it is easy to automate but if the

07:31

process is creative it can be very difficult to automate in many cases when we

07:41

do security testing we do not have source code so many security testing

07:51

techniques are black-box testing techniques and they do binary code

07:58

analysis byte code analysis instead of source code analysis

# CSE 4321 Security Testing (Part 2, Fall 2020)

00:05

next we look at a couple of security vulnerabilities we find

00:11

in a lot of practical applications I

00:19

believe many of you have heard about buffer overflow this is considered to be

00:29

one of the most common security problems especially in languages like C C++

00:38

basically we have a buffer overflow if we write beyond the capacity of an

00:47

array like data structure this would be a real example

00:55

of buffer overflow so this is because the index of an array begins at

01:04

zero so the last index of the array should be 9 if we write into

01:13

the tenth position we have a buffer overflow when a buffer overflows we

01:20

could have the program crash or we could have remote code execution or in some cases

01:41

the hacker could take complete control over a computer to some extent we

01:51

consider a program crash is lucky because in other cases the hacker could

02:00

do a lot more damage to the system this is a typical memory layout for

02:10

a program so we have text segment which basically includes

02:18

the code the instructions of a program and we have

02:24

the data segment which has the global data structures then we have the heap

02:40

and the stack the heap goes from low to high and stack goes from high to low so

02:54

when the heap meets the stack then the program runs

03:00

out of memory this is a very simple program in the

03:10

main method we call this method and this

03:20

is a command line argument and it is a string and in this method we create a

03:33

buffer that could hold six characters then we copy the string to the buffer we

03:48

also have another method that formats the entire hard disk

03:55

when we execute the main function we create a

04:00

frame for this function call and

04:12

when the main method executes the foo method we create another frame on top of

04:22

the frame for the main function call so

04:32

inside this frame for the Foo method we

04:38

allocate space for this buffer and we also save the return address so that

04:53

after we finish this function call we could come back to the main method but

05:16

what if we pass a very long string to this method so what

05:23

happens is that this function strcpy does not check if the buffer has

05:35

enough space to to hold this string so

05:45

when this string is more than six

05:49

characters the additional characters could overwrite the return

05:57

address if we construct this argument in a special way we could have this return

06:15

address overwritten

06:17

in a way that it points to this method

06:24

so when this happens after we finish this function call instead of going back

06:35

to the main function we come to this function and we execute this function

06:45

that would format the entire hard disk

06:49

SQL injection is another very common type of vulnerability in many

07:03

applications we have a login window where the user has to provide the

07:10

username and the password in order to use the application one

07:16

way to check the user name and password is that we construct a SQL statement

07:24

we try to find whether there is any account that has the same username and

07:33

password if we could find any

07:39

account that means the user can

07:42

access the application if you look at the code here basically

07:56

whatever username the user provides will be put here and whatever password

08:12

the user provides will be put here what if we put a username like this then

08:39

this username would be put into this select statement so it's basically

08:56

going to put it here what we have would be like this and we put the password here

09:34

if we look at this condition because we have this component and this part is

09:55

commented out so this entire condition is true that means the Select

10:13

statement would return all the accounts in the database and that means this

10:22

condition is true so we would be able to login even though we do not have a valid user

10:35

name and password

10:51

cross-site scripting is another very common vulnerability we find

10:58

especially in web applications basically the attackers inject

11:08

scripts that can be executed on victims machine they can use the injected script

11:17

to steal information for example from the cookies there are two types of

11:25

cross-site scripting reflected cross-site scripting sends the victim

11:32

a link that executes malicious scripts when the victim

11:39

actually clicks the link the second type of cross-site scripting is

11:45

called stored cross-site scripting so it basically posts malicious scripts

11:54

on a web forum so when the user gets on to the forum the script gets executed

12:03

and the execution of the script could steal sensitive information this

12:12

is an example of reflected cross-site scripting attack so we have a web server

12:30

called 123.com so if we access a webpage that does not exist on the web

12:40

server the web server would return a response page back to the client

12:55

browser and display this information so in the response page the server just

13:06

copy this URL and put it here so in this sense it reflects this URL back to

13:32

the client browser and this is a vulnerability

13:37

because a attacker could exploit this vulnerability by sending an email that

13:44

contains this link so when the receiver of this email tries to open this link so

13:56

a HTTP request would go to the server the server would

14:05

consider this page doesn't exist so the server would return the response page

14:15

where it tries to put this link in the response page so the client browser

14:29

receives this response page it would execute this malicious code because the

14:43

response page is coming from this web server so the client browser would

15:02

allow this malicious code to access the cookie maintained by this server but if we

15:16

look at the code it sends this cookie information to a malicious website

15:27

if we look at the three different types of vulnerabilities we have discussed the fundamental

15:39

problem is that we didn't do adequate input validation in this example we

15:51

didn't check how many characters we have in this string in SQL injection

16:03

we didn't check this user input before we put it into the Select statement this

16:15

user input contains some special characters that should not appear in the user name

16:23

over here the user input contain some malicious code which we

16:34

should not copy and paste into the response page

# CSE 4321 Security Testing (Part 3, Fall 2020)

00:02

next we discuss fuzz

00:04

testing a very widely used approach to security testing

00:14

basically fuzz testing uses random data to do testing for test oracles fuzz testing

00:24

just monitors for program crashes so basically if a

00:32

program crashes during a test execution we say that we have found a vulnerability

00:45

in the

00:46

program this approach is very simple

00:49

to apply and it can be applied to real-life large systems but the

00:57

disadvantage is that because we are doing random

01:03

testing it could be ineffective these

01:12

are the common types of inputs we use for fuzzing we could use very long or

01:27

completely blank strings we could use the maximum or minimum values

01:35

of integers or we just use zeros we could use special

01:43

characters or keywords that have special meaning and of course we also use a lot

02:00

of randomly generated inputs so in a fuzzing tool we have three major

02:14

components the first component is fuzz generator this component

02:22

basically responsible for random input generation there are two general

02:29

approaches which we'll discuss in more detail delivery mechanism is responsible

02:38

for delivering the test inputs we generate to the system that is being

02:50

tested we may have to create a file from the test inputs so that the

03:01

system under test could read the file we may have to set up some environment

03:09

variables we may have to send test inputs across a network to test the

03:18

system being tested we may have to convert the test inputs

03:29

into operating system events

03:33

such as mouse events keyboard events in order to test

03:39

the system under test

03:57

we also need a component to monitor the runtime behavior

04:49

of the system being tested during each test execution we can do local monitoring

05:26

we

05:27

could also do remote monitoring

05:40

one approach to generate fuzz inputs is that we take

05:46

one or more valid inputs and then we make random changes or heuristic

05:55

changes this approach requires little or no knowledge of the input structure because

06:06

we just make changes to a well structured input the changes could also include

06:16

some special values for example we could change an integer input to

06:24

the maximum value of the integer or the minimum value or some boundary values

06:35

for example we could fuzz a PDF reader by starting with one or more valid PDF

06:45

files then we try to mutate the files to create other PDF files the advantage of

06:56

this approach is that it does not require much knowledge about the input structure

07:09

the disadvantage of this approach is that the kind of inputs it could generate

07:17

depend on the initial inputs

07:22

generation based approach is to create inputs from scratch to do that

07:38

typically we have to use some kind of specification about the input structure

07:47

we could also make some random or heuristic changes at certain parts of

07:56

the input structure for example if we want to test a PDF

08:05

reader we could use the

08:14

specification of the PDF file format and then we generate random PDF files

08:22

based on the specification the advantage of this approach is that

08:34

it requires the specification of the input structure which may not be

08:43

available in some cases the advantage of this

09:06

approach is that it could be more effective because it takes advantage of

09:15

the specification so that it could produce inputs that cover different

09:22

types of inputs

09:35

people have developed approaches to improve the effectiveness

09:40

of fuzz testing one approach is that we could try to use information from the

09:58

source code to help us generate more effective test inputs so let's look at

10:09

this simple program we have a crash here and this crash is only triggered if

10:23

this condition is true and if we want to make this condition true then these four

10:37

conditions before this condition must be true as well

10:48

you can compute the chance for random testing to make this condition true and

11:03

the chance is very very small so what

11:12

white-box fuzzing does is that it tries

11:17

to systematically explore every possible path in a program so that in

11:24

the end we can find a test input that could trigger the crash so first we could

11:34

execute the program with a random input for example we first

11:41

execute the program with the input string good

11:46

when we execute this program with good

11:54

we know these conditions would not be satisfied we record the

12:06

path that is executed and the path would be the left most

12:19

path in this structure then what we do is

12:24

that we look at the last branching condition and we try to negate the last

12:34

branching condition if we want to make this condition true we know the

12:44

last character has to be the exclamation mark so

12:52

what we do is that we change this input

12:58

to this one then we could execute this path this one here but at the last branching

13:25

condition we will take this branch the true branch then we are going to back

13:38

it

13:39

up we try to negate this branching condition we try to negate this branching

13:52

condition the third character has to be D so

13:59

we change good to this one then we

14:16

execute the program we would execute this path this one this one this one

14:28

this one so here would explore

14:33

the other branch and then the last branching condition would be

14:55

false because the last character is not exclamation mark then we negate the

15:08

last branching condition then we change the input to this one this would allow

15:27

us to execute this path from here to here to here to here to here

15:49

so we can continue to

15:55

do this each time we try to negate a branch condition and that would allow us

16:05

to explore a different path in the end we would get to the path that would

16:13

trigger the crash which is basically the right-most

16:18

path as you can see here we would get the the inputs that is needed

16:27

to trigger that crash so basically this tree structure represents

16:39

all possible paths we have in the program and as you can see here the

16:53

white box fuzzing approach could be much more efficient than pure random testing a little

17:14

bit history of fuzz testing the original idea was proposed a little bit more than

17:23

20 years ago it is interesting to note it came out of a class project the PROTOS

17:33

project and the SPIKE tool apply fuzz testing to security protocols

17:43

the wide adoption of fuzz testing for security testing happened in 2005 the idea of

17:54

white box fuzzing was introduced in 2007 I believe this tool is developed by

18:05

Microsoft and this tool is developed by UC Berkeley next we recap what we

18:17

have discussed security has become a significant concern in software engineering

18:30

we discussed three common types of security vulnerabilities including

18:38

buffer overflow command injection and cross-site scripting

18:44

security testing is to break a system like a hacker but with a good purpose we

18:54

discussed fuzz testing it is basically random testing and it can be very

19:04

very effective because now we have so much computing power available so we

19:11

could try a huge number of randomly generated

19:16

inputs we discussed white box fuzzing sometimes people refer to that as smart

19:27

fuzzing it makes fuzzing more effective by using additional information

19:36

about the input structure and sometimes about the source code

# CSE 4321 Overview of Software Maintenance (Part 1, Fall 2020)

00:02

this course has two major parts testing and maintenance starting from this video

00:10

we move to maintenance first we give an introduction

00:16

to maintenance then we discuss different

00:21

process models they are used to organize different maintenance activities program

00:32

understanding is a important task of maintenance because we have to

00:38

understand a program first before we could maintain the program we discuss

00:48

configuration management it is basically about how to manage different versions

00:54

that we create during maintenance we also discuss management issues

01:02

how to maximize the productivity of a maintenance team maintenance is about

01:14

how to manage different changes we make to a software product after we deliver the

01:26

product we make changes to a software product for different reasons including

01:36

bug fixes new features environment adaptations

01:44

performance improvement and other reasons

01:50

maintenance can easily take 40 to 70 percent of the entire cost of a software

01:59

product what happens is that the more successful a software product is the

02:08

more time and effort we spend on maintenance

02:23

this shows the difference between software and program

02:32

basically documentation is what makes a program become a software product if

02:46

you do not have documentation you can not call what you have is a software

02:52

product there are two types of documentation these documents are created for the

03:05

developers so these are typically referred to

03:11

as technical documents we also have user documents these documents show the user how

03:23

to install and how to use the product

03:39

so what's the difference between maintenance

03:42

and development when we do development we build a

03:49

new system from scratch maintenance is different when we do maintenance we have

03:56

to work within the parameters and constraints of an existing system

04:06

this means that typically we have more restrictions when we make decisions during

04:14

maintenance than during development because of that maintenance could be

04:23

more challenging than development in particular if you consider it is

04:32

typically much more difficult to add a new room to an existing building than adding the

04:38

room when we first build the building before we

04:49

could maintain a system we have to first understand the system some

04:58

important questions we want to understand how to accommodate the change we want to

05:06

make what is the potential impact of the change on the rest of the system what skills

05:21

and knowledge are required before we could maintain the system

05:34

so why we do maintenance one reason is to provide

05:40

continuity of service we make changes to fix bugs to recover from failure to accommodate

05:51

changes in the environment a second reason is to support required upgrades we

06:02

make changes to keep up with government

06:06

regulations to maintain competitive edges another reason for maintenance is

06:17

to support user requests for improvements including new features

06:26

performance improvements customization for new users we do maintenance also to

06:36

facilitate future maintenance work including refactoring document updating

06:47

refactoring is basically about moving code around to improve the quality of the

06:55

code so that it is easier to make changes in the future we will discuss

07:03

refactoring in more details these are some basic properties about software

07:20

systems continuing change software systems are very dynamic we make a lot

07:28

of changes to a software product even after its release it is different from

07:38

other products for example we typically do not make additional changes

07:49

after a movie is released increasing complexity as the time goes a software

07:59

system becomes more and more complex continuing growth a software system will

08:09

have more and more features in order to make the user happy in

08:18

general the more the user uses a software product the more the user may

08:26

want from the product declining quality the quality of software product will go down

08:36

over time this is particularly so after multiple people have touched the product have

08:46

made changes to the product these are the major activities we do during

09:00

maintenance first we want to identify what change we want to make and we want to

09:12

justify why we want to make the change then we want to understand the existing

09:22

system we want to understand how to make the change what components must be

09:33

changed and what is the potential impact of the change

09:41

next we are ready to actually implement the change and after

09:55

implementation we do testing and make sure the change is actually implemented

10:05

correctly configuration management is about how to manage the changes we make

10:14

each time we make a change we create a new version configuration

10:24

management helps us manage the different versions we create during maintenance a

10:42

maintenance team has to be managed effectively so that the team can be

10:50

productive and can do a good job on maintenance

# CSE 4321 Overview of Software Maintenance (Part 2, Fall 2020)

next we discuss process models how to organize different maintenance

00:12

activities these are the major models

00:24

for development these days iterative models are becoming very popular these

00:33

models divide a project into multiple iterations each iteration is like a mini

00:45

project the main advantage of iterative models is that it allows us to get

00:55

user feedback at the end of each iteration so that we could

01:04

make adjustments in the next iteration this model for maintenance is similar to

01:26

the code and fix model for development so basically we find a problem and we

01:36

just fix it this model is actually ad hoc and it is not very well defined

01:51

this is a more well-defined process we first propose the changes we want to

02:03

make we get management approval we implement the changes

02:12

we get a new version we use the new version and evaluate the results and if

02:23

needed we propose new changes this model emphasizes the management's

02:36

decision so whenever we propose a change we have to get approved before we

02:47

actually implement the change this is a more elaborate model so basically it

03:02

takes each change as a new feature development first we identify the

03:11

change and we submit our change requests we do requirements analysis we get

03:25

change request approved then we do task scheduling we do design and analysis design

03:45

review before we actually modify the code to implement the change and after

04:00

we implement the change we review the change implementation we test we update

04:11

documents we do standards auditing to make sure we follow the coding standards

04:20

we do user acceptance testing and post installation review of

04:28

the changes we have implemented before we complete this change in the middle of

04:37

the process we could go back to previous steps as needed this model is useful

04:50

for systems that we want to have very high confidence we want to make sure each

05:02

change is implemented correctly that is important for systems that for example

05:13

operate in a safety critical domain this model is similar to the iterative model

05:27

for development so basically we analyze the existing system we propose the

05:36

changes we want to make and then we implement the changes we repeat this process as

05:47

needed this is some statistics about where maintenance effort is spent

05:59

non discretionary maintenance refers to changes that we must make for the system

06:08

to continue operation discretionary maintenance means optional maintenance

06:18

so basically changes we may choose not to implement this data shows

06:32

we spend more time on discretionary maintenance than non-discretionary

06:39

maintenance this shows the difference between maintenance and development

07:04

so maintenance spends more effort in analysis

07:10

specification and design development spends more effort in

07:18

implementation testing the reason is that when we do maintenance in many cases

07:29

the actual modifications we have to make are very small but

07:36

before we are able to make those modifications we have to spend time and

07:44

effort to understand the system to understand the changes we have to make

# CSE 4321 Overview of Software Maintenance (Part 3, Fall 2020)

00:02

next we discuss program understanding we have to understand a system before we

00:11

could maintain the system so what to

00:21

understand about a system first we want to understand the business domain

00:28

the system operates in for example if we are trying to understand a financial

00:37

system we have to understand how financial people conduct business

00:43

transactions we could get domain knowledge from the documents by

00:51

talking to the end users or by reading the source code we also want to

00:59

understand the execution effect basically how the system behaves at

01:10

runtime for any system we want to understand we ask one

01:20

important question

01:33

what input the system takes and what output

01:38

the system produces we also want to understand how data flows from one point

01:47

to another in the program and how control flows from one point to another

01:56

in the program at runtime we also want to understand the core business logic how

02:10

the input is transformed step by step to the output the system produces cause

02:19

effect relation is basically about the dependency between different components

02:28

how they effect each other this relation is very important for maintenance

02:37

because it helps us to understand the potential impact of the change we make what

02:48

components could be affected when we make a

02:53

change we also want to understand the product environment relation

03:01

basically how the product talks with its environment

03:15

to understand a system we could first read its documentation good

03:22

documents should be well-written easy to read and should give us the most important

03:33

information about a system

03:46

we could read the source code and reason its

03:54

possible behavior at runtime this is basically static analysis so if we

04:04

compare source code to documentation the source code is more precise because

04:12

documents are written in English different people

04:17

reading the same text may have different interpretations in

04:24

contrast the semantics of the source code is precisely defined so there's

04:33

only one way to interpret the source code but the source code could be

04:42

overwhelming because it contains a lot more details than the documents we could

04:51

also try to understand a system by putting the system into action this is

04:59

basically dynamic analysis so we run the system and directly observe how the system

05:08

behaves there are different strategies we could use when we try to understand a

05:27

system we could go top down basically this means we could first try to

05:37

understand the architecture of a system in terms of the major components

05:44

and how the major components talk to each other before we try to understand

05:52

the details of each component we could also go bottom up

06:02

so we first try to understand the smaller components then we try to put

06:09

the smaller components together to understand bigger components until we

06:16

understand the entire system and we could also combine the two approaches

06:24

for some parts of the system we could do top down for other parts of the

06:33

system we could do bottom up I think in practice most people use the mixed

06:44

approach because in some cases top-down approach is better and in some cases

06:53

bottom-up approach works more effectively these

07:00

are some major factors that could affect how we understand an existing system

07:18

expertise in terms of domain knowledge and programming skills the more domain knowledge

07:26

we have the more programming experience we have the

07:31

easier for us to understand the existing system program structure is an important factor a

07:50

good structure should be modular so basically

07:55

it means different components in the system should be relatively independent

08:02

and each component should have a well-defined interface we should reduce the level of nesting

08:14

the more nestings we have the more difficult a system

08:19

to be understood documentation can really help good documents should be easy

08:29

to read should contain accurate information and should be up to date with the system

08:38

implementation it is very important not to forget updating documents after we make

08:53

changes to a system implementation we want to have good coding conventions in particular

09:09

we want to give meaningful names to

09:13

variables methods and classes for example a

09:18

method should have a name that indicates what the method is supposed to do we want

09:29

to have quality comments inside the

09:32

source code we do not want to write comments just for the purpose of writing

09:41

comments each comment we write should contain some useful information that could

09:50

help people to understand the code we also want to make good use of

10:00

indentation and spacing to improve the presentation

10:06

of the program a good presentation can make the program much easier to understand

10:26

reverse engineering is an important approach to

10:30

program understanding reverse engineering is basically about abstraction it tries

10:41

to create more abstract system representations from the source code so

10:49

that we could focus on the more important aspects instead of

10:56

every detail that is contained in the source code three types of abstraction

11:10

are used in reverse engineering function abstraction

11:17

tries to abstract a function in terms of what it does instead of how so

11:30

basically it tries to focus on the input and output of a function instead of

11:40

the implementation details data abstraction tries to abstract the data

11:47

structure in terms of the operations that we could perform on the data

11:56

structure instead of the specific representation details in the data

12:01

structure process abstraction is used to understand systems that involve multiple

12:11

processes it tries to focus on how the different processes communicate and

12:21

synchronize with each other instead of internal computation details of each

12:29

process so why we want to do reverse engineering we could use reverse engineering to

12:43

improve or provide documentation for example if a design decision is not

12:52

captured by a design document we could use reverse engineering to recover that

12:59

design decision and use that information to update the design documents reverse engineering

13:12

could cope with complexity this is because reverse engineering is about

13:23

abstraction abstraction can reduce the amount of

13:29

information we have to process reverse engineering can help to identify reusable

13:39

components sometimes different components are different

13:45

only in terms of implementation details when we look at these components from a more

13:54

abstract perspective they actually do the same computation reverse

14:02

engineering can also help migration between different platforms this is

14:13

because we could recover from a platform specific implementation design

14:23

decisions that are platform independent then we could implement those platform

14:31

independent design decisions on a different platform reverse engineering

14:39

also provides a different way to look at the subject system we try to

14:46

understand we could do reverse engineering

14:57

from implementation to design so basically we try to recover design

15:06

decisions from the source code we could

15:15

also do reverse engineering from design to specification

15:22

so basically we try to recover specifications from the design documents

15:31

just to comment that if we do forward engineering we would go from top down reverse

15:41

engineering is going the reverse direction

15:46

so basically bottom up in forward engineering we add details more and more

16:03

details into the system implementation reverse engineering is trying to abstract

16:12

away details from the implementation

# CSE 4321 Overview of Software Maintenance (Part 4, Fall 2020)

00:02

next we discuss configuration management

00:13

configuration management is a very important

00:16

component in the management and maintenance of any

00:22

large project one way to put it is that

00:30

if we could have only one tool to manage our project

00:38

configuration management is probably the tool we want to have

00:46

consider that we get a bug report from a customer

00:51

to fix the bug we would have to have the

00:56

same version of the software as the customer does

01:03

this is because otherwise we would not be able to

01:10

reproduce the bug configuration management

01:16

can help in this case it allows different

01:21

releases to be made from the same code base in particular

01:28

it allows to go back to any previous

01:33

release we made configuration management also supports

01:39

effective teamwork so basically it allows many people

01:47

to work on the same files at the same time

01:56

it also supports accounting and auditing they are important

02:03

from the project management perspective

02:06

these are the major activities performed by configuration

02:15

management first it identifies every component

02:22

in the system and every change made to the system

02:28

it also helps to exercise control

02:32

over the way the changes could be made

02:37

it supports accounting and auditing

02:44

so basically it could record and document

02:50

all the activities that are taking place on the project it also helps

02:57

us to inspect the different activities and the different system

03:04

states this is very important from the

03:11

project management perspective this is because if something

03:17

goes wrong this allows us to go back

03:21

to look at everything that has happened so that we could

03:26

figure out what was going on another benefit

03:37

of configuration management is that if the developers

03:43

know what they do is being recorded

03:47

they could be more responsible this is the big picture

03:57

configuration management is an important component of

04:02

any project management i also want to comment

04:09

on this component sometimes people refer to

04:14

this as a bug tracking system or a bug reporting system

04:27

this is also a very important component because

04:32

for any bug that is reported we want to

04:36

keep track of its status and make sure it is actually

04:43

fixed as we discussed before

04:54

in the life cycle of a software product we make

05:02

a lot of changes each time we make a change to

05:09

an object we create a new version of the object

05:16

so over time different objects could have different

05:23

versions when we make a release of the entire system

05:35

we have to put different versions of different objects

05:41

together for example when we make the first release we may include

05:48

the first version of each object

05:53

when we make the second release we may include version number one of

06:02

the first object and then version number two

06:10

for every other object and for our release 3.0 we may include

06:22

the second version of object one the third version of object two

06:29

the fourth version of object three and the second version of

06:41

object four if we look at individual

06:50

objects a object

07:01

could have many different versions in this example the different

07:07

versions could be put into a tree structure so this

07:15

structure basically represents the version history of

07:22

this object build is one of the most

07:33

frequently performed operations when we do development and

07:39

maintenance this is because whenever we make a change we have to

07:45

recompile the system rebuild the system

07:52

when a system is small it does not take much time

08:00

to build but if a system has a lot of files

08:06

in a real-life project we could easily have

08:10

hundreds of and even thousands of files

08:15

to rebuild the system from scratch it could take a lot of

08:26

time sometimes maybe a couple hours to finish

08:32

building the system from scratch could be time consumingso to be more efficient

08:52

we want to do incremental building the idea is that

08:58

we only want to rebuild objects that have changed

09:03

or that have been affected by a change

09:09

so that we do not have to rebuild everything every time

09:15

we make a change to the system this can significantly

09:23

speed up the building process

09:26

when we build we have to make sure

09:31

the different versions of the different files can

09:36

work together are consistent with each other so

09:44

to do incremental building we have to know

09:48

the dependency relation between different objects

09:53

and this is what makefiles do

09:59

we typically use a make file to define

10:03

the dependencies between different objects so that when we do

10:09

incremental building we know what objects have to be

10:18

rebuilt configuration management supports

10:24

change control before we make a change

10:33

first we have to decide if the change should be made

10:39

is it a valid change does the cost outweigh

10:45

the benefit do we have any potential risk

10:52

keep in mind that when we make a change

10:55

we are at risk to break the existing system

11:01

so in some cases we may decide not to implement a change

11:07

if the risk of breaking the system is too high

11:16

after we decide to make a change then we need to manage

11:22

the actual implementation of the change in particular

11:30

we want to record the change and we want to monitor

11:36

the progress it is important to record the change

11:41

because if something goes wrong we could

11:45

come back and inspect the change that would help us to figure out

11:52

what was going on and after we implement

12:00

the change we have to do adequate testing to make sure

12:05

that the change is implemented correctly this

12:12

is an example change request form in the form

12:18

we want to identify the name of the system that is being

12:23

changed we want to identify the version number revision number

12:32

the date who is requesting the change

12:35

a summary of change why we want to make the change

12:42

the software components that

12:44

require changes and documents that

12:51

require changes we also want to include an estimate

12:57

about the cost to implement the change

# CSE 4321 Overview of Software Maintenance (Part 5, Fall 2020)

00:01

next we discuss management issues basically how to effectively manage a maintenance

00:10

team when we manage a team we want to maximize their productivity we want to

00:20

do good personnel management we want to choose the right people we want to

00:29

motivate the team when people are motivated they can do a much better job

00:37

we want to keep the team in the loop we want to allocate adequate

00:45

resources so that the team have the needed resources to do the job

00:54

we also want to choose a good organizational mode we need to decide

01:02

whether we combine or separate the development and maintenance team when we

01:11

combine the two teams it means the same people would be responsible for both

01:19

development and maintenance we want to decide between module ownership and

01:30

change ownership depending on which model works better for our project these are

01:42

some good approaches to motivating the team financial rewards promotion

01:55

could always help in addition we want to provide technical supervision and

02:08

support especially for inexperienced staff because we do not want people to feel

02:18

lost when people are lost they cannot do the job well we also want to

02:29

rotate between maintenance and development this could make the work

02:36

more interesting and it could also help to build up the resume which is important

02:46

for people's career development recognition

02:56

is a very effective way to motivate people if someone has done a good job

03:04

you want to send out an email to the team to acknowledge that we

03:13

also want to provide opportunities for one to grow professionally education

03:28

and training can get people technically ready for the job can make them more

03:40

productive educational and training should be

03:46

at the heart of an organization not a peripheral activity different ways for

03:57

education and trainning you could send people to university for advanced education or

04:07

you could send people to conferences and workshops this is a very good approach

04:15

to keep people up-to-date with the latest developments you could also

04:30

provide hands-on training this is one way to organize maintenance work in

04:49

this model each component has an owner the owner is responsible for all the

04:58

changes in this component if we have a change that has to change multiple

05:11

components then the owners of those components have to work together to

05:19

implement the change the main advantage of this model is that over time the

05:27

owner develops a very good understanding about the component the owner is

05:34

responsible for but this model has a couple of disadvantages first in this model

05:44

no one is responsible for the entire system this is because each owner is

05:52

only response for an individual component the workload

05:59

will not be evenly distributed because some components may need to be changed

06:07

more frequently than other components also in this model it could be difficult

06:15

to implement changes that require collaboration of multiple components this

06:24

is because we don't have a single person responsible for the change this is a

06:37

different way to organize maintenance work in this model each change has an owner

06:46

the owner is responsible to implement the entire change end-to-end if the

06:55

change involves multiple components the owner is responsible for changing all

07:03

those components this model has a couple of advantages in particular

07:10

this model helps to make sure the integrity of the change this is because

07:25

one owner is responsible for the entire change in this model changes could also

07:34

be implemented and tested independently this is because the owners of different

07:43

changes could work at the same time this model also has some disadvantages

07:53

first the training of new people could be difficult the reason is that if a

08:01

change involves whatever components the the change owner has to have good

08:09

knowledge about all those different components otherwise the owner would not

08:17

know how to change those components in

08:21

addition in this model the change owners do not have long lasting responsibilities

08:30

in the sense that next time the change owner

08:34

may be assigned with a very different change so in this case a change owner

08:42

is likely to make short-term decisions because the owner just wants to make the

08:51

current change work this is different in module ownership when the module owner

09:01

makes the change the owner does not only consider the current change but also how

09:10

the decision is going to affect the future changes

09:24

next we recap what we have discussed maintenance is a very important stage in

09:40

the software lifecycle it must be managed efficiently the

09:50

fundamental difference between maintenance and development is that

09:59

maintenance has to work with the constraints of the existing system

10:15

maintenance can be more challenging than development a unique challenge is that

10:22

we have to understand an existing system before we could maintain the system

10:36

maintenance is about change management how to manage how to control

10:43

the changes we make to a software product after the product is released

10:52

it is important to keep in mind we do not only maintain the code we also need

11:01

to maintain the documentation we need to make sure the documents up-to-date

# CSE 4321 Version Control (Fall 2020)

00:02

in this video we discuss a new topic version control

00:08

first we give an introduction

00:11

look at the big picture then we introduce two important

00:16

concepts product space version space they

00:22

are used to represent the different software

00:26

objects and the different versions the software objects

00:33

could have in a software product we talked about

00:39

how product space and version space talk to each other

00:46

we also discuss intensional versioning a very common approach

00:52

to managing the different versions

01:03

we discussed configuration management before

01:09

which is a very important tool for us to achieve

01:16

a well-defined process model for

01:20

software development and maintenance

01:31

configuration

01:32

management provides support for both management and

01:37

development for management configuration management

01:42

could be used to record all the activities that take place

01:48

on the project

01:50

that can be very useful in case that we want to

01:56

review what is going on in the project

02:01

for development configuration management allows multiple developers

02:09

to work on the project at the same time and

02:13

it allows us to make different software

02:18

releases and to go back to any previous

02:26

release

02:30

version model is at the core of any version control

02:34

system it defines what objects to be versioned

02:41

how to identify and organize the different versions

02:49

it also defines operations that could be used to

02:54

to retrieve a existing version and to construct

03:00

new versions

03:02

three important concepts in a version model product space

03:12

is used to

03:14

describe represent different software objects and

03:18

the relationships between the different objects version space is used

03:25

to represent different versions of individual software

03:31

objects versioned object space basically

03:36

combines both product and version space

03:47

next we discuss each of these three concepts in detail

03:55

first what is product space product space

04:03

is used to represent the structure of a software

04:09

product in the product space we do not take into account

04:16

the different versions a software object could have

04:23

from another perspective in the product space each

04:28

software object only has one version

04:33

we typically use a structure called product

04:37

graph to represent the product space in the

04:43

graph structure each node represents a

04:48

software object each edge represents a relationship

04:54

between different software objects

04:57

software objects are artifacts we create during

05:11

development or maintenance

05:26

in general there are two types of software objects

05:32

source objects are artifacts we create directly

05:37

as a software engineer

05:40

examples of source objects include

05:44

source code files we write design documents we write

05:50

test cases we create

05:57

derived objects are not

05:59

directly created by the developer instead they are created

06:02

using some tools from some source

06:07

objects for example when we compile the source

06:12

code files we create compiled objects those compiled

06:20

objects are derived objects each software

06:25

object has a unique identification this

06:29

is needed so that we could identify

06:34

the different software objects this is similar to what we do

06:38

in real life for example in school we have to assign

06:46

each student a unique identifier so that we could

06:51

manage the different students

06:56

a software object may or may not have a

07:02

internal structure represented in the version control system

07:06

for example a program file could be represented as

07:11

a text file in this case we do not represent

07:16

the internal structure of the program or we could

07:23

represent a program file as a syntax tree in this case

07:29

we represent the syntactical structure of

07:40

the program

07:41

there are two types of relationship

07:43

that can be captured in a version control

07:47

system composite relationship

07:50

indicates which objects are composed of

07:55

which objects dependency relationship indicates

08:02

which objects depend on which objects

08:06

life cycle dependency exists between artifacts

08:13

that are created at different life cycle stages for example

08:21

design documents depend on requirement documents this is because

08:31

design decisions are made to implement the requirements

08:36

specified in the requirement specification document

08:44

import/include dependencies

08:49

exist between different

08:50

components for example one java class may import

08:56

another java class in this case we have import

09:01

dependency between these two classes

09:05

build dependencies exist between compiled code and

09:11

source code so basically the compiled object

09:17

depends on the source object that is used to generate

09:24

the compiled object

09:25

this is a example product graph

09:35

in the graph each node represents a software

09:40

object each edge represents a relationship

09:44

between two different objects

09:47

this dashed arrow represents a dependency relationship

10:01

this solid edge represents a composition

10:07

relation for example if we look at this

10:12

root node it indicates the entire system

10:17

is composed of these four different

10:24

objects and if we look at this dashed

10:37

arrow it indicates this object depends on this object

10:49

what is version space

10:51

version space is used to represent the different

10:56

versions of individual objects

11:00

each version is basically a state of an item

11:04

that is evolving versioned item

11:07

is an item that we put under version control that basically

11:14

means we keep track of

11:18

the version history of the item which includes

11:22

all the different versions we create

11:26

for this item versioning can be applied at

11:31

different levels of granularity we could version

11:35

the entire software product or we could version

11:42

individual components of a software product

11:46

for each version we have to have a unique identification

11:57

so a version identifier has two

12:01

parts the first part is that

12:05

it has the object identifier that

12:09

can be used to determine whether two versions

12:12

belong to the same item then we have the

12:19

version identifier that is used to uniquely identify a particular

12:24

version for the same versioned item

12:36

one technical problem in a version control

12:39

system is how to store the different versions

12:44

as efficient as possible the reason is that we could create

12:51

a lot of versions in the life cycle of a

12:57

software product and in many cases

13:01

we make small changes the new version

13:08

is largely the same as the old version so we do not want to

13:18

store the two versions separately

13:22

because they are largely the same a better approach is to

13:29

store the differences between the old version and

13:34

the new version one way to do that is we explicitly

13:49

store the differences for example here for

13:55

these two versions v1 and v2 we store

14:01

what is in v1 but not in v2 and we store what is in v2

14:12

but not in v1 for what is

14:25

in both v1 and v2 we only store one copy

14:35

by doing so we could

14:40

reduce the storage requirement this is particularly so

14:48

if the

14:49

the intersection between v1 and v2 is big

14:55

a different approach is that we could store the sequence of

15:01

operations that are performed to create the new version

15:13

so if we want to get the new version we just take

15:19

the old version and redo the sequence of

15:24

operations

15:36

in general there are two strategies

15:41

to manage the different versions extensional

15:46

versioning in this strategy each version is

15:50

explicitly represented in the version control system

15:56

so if a object has five versions

16:01

then we would have five different versions explicitly

16:04

stored in the version control system and each version

16:17

is by definition immutable that means if we make a change

16:23

to a version then we get a new version

16:28

the second strategy is called intensional versioning

16:33

in this strategy versions are not explicitly

16:39

represented and stored in the version control system

16:43

instead different versions are created

16:52

as needed for example conditional compilation can be used

16:59

to construct different versions of a

17:02

source file based on for example some environment

17:16

attributes

17:18

these are two important concepts in version control

17:24

revision represents a version that is intended

17:29

to replace its predecessor so if

17:35

for example we fix a bug then

17:40

the new version that has the bug fix

17:44

is intended to replace the previous version and similarly

17:50

if we add a new feature

17:54

the new version that has the new feature is intended

17:59

to replace the predecessor the previous version a variant

18:07

is a version that is intended to coexist with the

18:15

previous version for example if we have

18:20

a software product that

18:23

supports both windows and mac then

18:29

the two different versions the windows version

18:31

and the mac version they are supposed to coexist

18:39

to exist at the same time

18:48

the different versions could be organized in different structures

18:53

they could be organized in a sequence or in a tree structure or

18:59

in a graph so

19:05

a tree structure allows multiple versions to

19:09

coexist

19:11

and the graph structure allows different versions

19:15

to coexist and then get merged together

19:23

the version structure could be hierarchical

19:32

so that means we could create different levels at each

19:38

level we could have sequence tree or graph structure

19:46

and the different levels could merge

19:50

this is similar to a file system where we could create

19:58

different folders

20:05

just want to make a comment sometimes we want to merge two different versions

20:14

for example if in this example if v2 is

20:20

created to fix a bug v3 is created to fix

20:28

another bug we could merge the two different versions

20:35

so that we could have a version that includes

20:40

both bug fixes

20:52

we could also represent the different

20:56

versions in terms of the changes

21:01

each version should contain in this

21:05

example for example v1 is created

21:10

by including changes c1 c2 c3

21:16

and if we include changes c1 c2 c3

21:26

c4 we could create version number two

21:42

we have discussed product space and version space we could

21:49

integrate product space and version space using

21:54

a model called and/or graph

21:59

in this graph we have two types of nodes

22:03

and nodes represent composition and or

22:12

nodes represent different versions

22:19

in this model both objects and configurations

22:24

can be versioned we will see examples of

22:29

the two different cases

22:34

in this example

22:35

the circle represents a or node

22:41

the box represents a and node so

22:48

if we look at this node this is a and node

23:03

basically it means if we make a release

23:07

we have to include all the child nodes of this node

23:15

this is a or node so basically

23:19

the child nodes represent different versions

23:24

of this

23:25

node or the object represented by this node

23:31

so when we make a release we can only choose

23:36

one of the different versions keep in mind

23:47

whenever we make a release each object

23:52

could only have one version we cannot have multiple versions

23:58

in a particular

24:00

release

24:10

in this example we do not

24:13

only version the individual components meaning that

24:18

we keep track of different versions for each

24:22

object we also version the entire system

24:28

so basically we keep track of the different versions of the entire

24:35

system

24:36

in version 1 we include all the five objects

24:48

in version two object c is

24:53

removed so each version of

24:57

this root node represents a configuration of

25:03

the actual objects a configuration is like a folder

25:10

in the file system versioning a configruation is similar

25:15

to keeping track of different versions of a folder

25:20

as mentioned before

25:34

intensional versioning does not represent each version

25:44

explicitly instead it creates new versions

25:50

on demand based on some properties specified

25:56

by the user

25:59

a software product could have many objects

26:04

and each object could have many versions when we make

26:13

a product release we pick one version for each

26:19

object in the product

26:23

so when we have many objects and many versions we could

26:28

have a lot of combinations to consider and not

26:34

every combination is valid for example

26:44

if we have a class a it uses

26:50

a method in class b but the method is

26:54

only available in class b after version 5.

27:00

so that means this method was added to class b

27:05

at version 5 that means if we make a

27:11

product release if we include class a then the version of class b

27:20

has to be at least version 5 that is because

27:27

otherwise the method would not be available

27:31

in versions before version 5.

27:37

this is what we refer to as consistency control

27:47

we want to make sure every new release we construct is consistent

27:54

configuration rules are used for this purpose to define what

28:01

combinations of the different versions are

28:06

valid and

28:10

very important if we create a new version that has

28:15

never existed before we have to be very careful we want to

28:22

perform adequate testing to make sure the new version is

28:28

consistent and

28:31

work as designed

28:37

this framework shows how intensional versioning

28:41

works we have two types of configuration rules

28:47

this one represents the user specified configuration rules

28:52

this represents some built-in configuration

28:58

rules this is the

29:01

versioned object base sometimes we refer to as

29:07

code repository so basically it includes all the software objects

29:16

and all the versions we have created for those objects

29:26

so the configurator takes as input these

29:30

two types of configuration rules and it goes to the

29:36

code repository and constructs

29:40

a version as requested by the user

29:51

the built-in rules are basically

29:53

rules that apply to all the

29:56

systems and cannot be changed by the user for example

30:05

whenever we make release we could only pick

30:14

at most one version of a software object this

30:20

rule applies to every system then we have user-specified

30:27

rules which basically define what version

30:33

the user wants to create

30:45

these are some example configuration rules

30:49

this one indicates we want to select the latest

30:56

version this one specifies a particular

31:01

version this one specifies that we want to

31:05

choose a version that works for

31:09

unix x11 and oracle

31:20

this indicates

31:21

we cannot have a version that works

31:24

for this combination of operating system and

31:29

windows system

31:33

this indicates we want to select these three change

31:39

sets this indicates that if we want to select

31:45

c2 then we must select c1 because c2

31:53

depends on c1 this indicates

32:01

we can only choose one of the three so this is the exclusive

32:09

or operator

32:22

the configurator basically has to

32:26

evaluate the configuration rules including

32:31

both built-in rules and user-defined rules

32:36

and it produces as output a version that

32:46

satisfies the user requests

33:02

these are some major configuration management

33:06

tools this is considered to be

33:11

the first version control system was developed in 1972

33:20

CVS was very popular I used

33:24

this system when I was in college Subversion

33:31

was developed to replace CVS

33:36

ClearCase is a commercial

33:39

version control system it is designed to support large

33:51

scale distributed software development

33:55

these days GitHub BitBuckets are really

34:02

becoming popular

34:11

to recap what we discussed version control is the technical

34:16

core of any configuration management system we discussed

34:23

product space and version space they are used to represent

34:28

different software objects and different versions

34:34

they also capture the relationship between different objects and

34:41

between different versions

34:43

three important technical problems in version control

34:53

how to represent the product space how to represent the version space

35:02

and how to store different versions efficiently

35:10

we could create a lot of versions and we want to minimize

35:18

the storage requirements how to present the user

35:24

a consistent view so when we make a product

35:30

release we want to make sure the different versions

35:35

of the objects we include in the product release

35:43

can actually work together

# CSE 4321 Code Review (Fall 2020)

00:02

in this video we discuss the topic of code review first we give an introduction

00:10

to the topic what is code review why we want to do it

00:16

then we discuss how to actually conduct code review we give some tips that we

00:24

could use in practice to do better code reviews we also discuss some tools that

00:33

are available to support code review what is it basically code review is to

00:48

inspect the source code in a systematic manner to make sure the code is of good

00:57

quality when we do code review we want to detect faults that may exist in the

01:05

source code in addition we want to identify

01:20

opportunities to make the code easier to understand and easier to maintain the

01:31

reasons for code review first code review can help to detect faults and

01:39

make corrections early in the development process so typically we have

01:46

multiple people look at the code during the code review so

01:52

they could find issues that the developer cannot find it can help to

02:01

improve code structure because people can give you suggestions about how to

02:08

better structure the code based on their experiences and from different

02:19

perspectives it can also help to enforce coding

02:26

standards because people actually look at your code they will check if the

02:33

coding standards are followed in the code code review is a good opportunity to

02:41

spread knowledge among different team members this is a good training

02:47

opportunity for new people in addition it is very important if the original

02:56

developer leaves the project another very important reason for code review is

03:05

that if the developers know the code will be reviewed they will work harder they

03:14

will write better code this is kind of human nature so in this respect

03:23

it does not matter how many issues code review actually finds if we

03:31

have the code review process in place it will help to improve the quality of

03:44

code so when do we do it and how frequently we want to do it we don't want

03:57

to do code review too soon the reason is that code review involves multiple

04:05

people it is very expensive so when the code is not stable yet it is not

04:13

efficient to have multiple people involved you do not want to do it

04:20

too late because if it is too late it may be too expensive to make any

04:28

corrections typically we want to do code review after unit testing has been

04:38

performed and after basic features have been tested and we could do it weekly or

04:47

we could do it after each major feature is implemented code review is a place

05:07

to discuss and learn from your fellow developers it is

05:14

not a opportunity to criticize people and it

05:19

is not a opportunity to show off who is a better programmer these are some potential

05:30

misuses of code reviews if code review is not managed effectively it could be a

05:41

big waste of time and effort the reason is that code review is very expensive

05:50

because it has to get multiple people involved the time and effort they spend

05:58

on the review are time and effort they could spend on other project activities

06:09

if code review is too harsh it may destroy the confidence of the developer

06:20

this is particularly so for the developer who is not very experienced if

06:26

ego or politics is involved in the review process

06:34

it could also create social problems

06:48

we could do the review informally

06:51

over the shoulder basically means if someone passes by your office you could

06:59

stop the person and ask if he could give you suggestions you could also email

07:08

your code to colleagues and ask for comments and suggestions or you could

07:17

use some software tools to help you to look at the code we could also do

07:27

formal code review in this case we have a well-defined process we have physical

07:38

meetings so the participants will meet in person the participants have to

07:46

do their homework before the review meeting they have to come prepared

07:56

we also have to document the results of the review meeting so that we could

08:02

verify that the comments made in the review meeting are actually addressed

08:16

this is one possible process to manage code review activities first we do planning

08:35

we prepare the materials that need to be reviewed we identify

08:41

participants who will participate in the review process and we find a meeting place

08:51

then we give an overview to the participants we want to educate the

08:57

participants on the materials they need to review and we want to assign

09:04

responsibilities to each participant then the participants prepare for the

09:13

review meeting they actually look at the materials that need to be reviewed and

09:23

then they get ready for the review meeting so

09:33

in the review meeting people make suggestions about possible faults that

09:42

may exist in the code and they also suggest opportunities that we could

09:49

take to better structure the source code to improve the design of the source

09:57

code then the developer go back to address the comments received from the review

10:07

meeting then we have to have a follow-up we want to verify that all the comments are

10:16

actually addressed I want to make a comment for any

10:31

meeting follow-up is very important if we do not follow up then whatever we

10:40

discussed in the meeting may not get implemented in that case the time and

10:48

effort that we spent would go wasted this is a simplified process in this

11:00

process we only have three steps in the first step we set up the review group we

11:08

invite people to the review process we make the materials available and we

11:16

ask the participants to get prepared then we have the actual review meeting in

11:28

the meeting the leader opens with a short discussion typically the leader would

11:36

clearly specify the goals of the meeting and the rules the participants

11:43

have to follow then the reader explains the code what the code is supposed to

11:54

accomplish what requirements it implements and what documentation it

12:02

affects then the participants ask questions make comments and give suggestions

12:14

then the developer responds explains the logic discusses the technical problems and

12:27

the possible choices to address the technical problems and then we want to have

12:44

a follow-up to verify the comments are actually

12:49

addressed what people we want to invite we want

13:03

to have a review leader this person should be a technical authority should

13:11

be experienced and should have a good personality we want to have a recorder

13:26

who keeps the meeting minutes keeps a written record for the meeting

13:36

we want to have a reader who reads the code that is being reviewed this

13:43

person could be the developer or a different person we want to have

13:54

an architect who has a good big picture we want to have some one in a similar

14:02

position as the developer we want someone in the middle

14:08

and new people who just come to the project in general we want to have a

14:21

balanced mix of the participants so that they could contribute from different

14:30

perspectives code review is a technical matter we do not want to invite

14:39

non-technical people and system testers because this is not the best way of

14:51

using their time we do not want to invite management this is because people

15:02

could get defensive when management is present

15:20

so what to look for during code review we look for programming mistakes

15:31

basically faults that may exist in the code

15:41

we also look for incorrect assumptions and misunderstanding of

15:50

requirements it is important to note that incorrect assumptions and

15:58

misunderstanding of requirements can be very difficult to find by the developer so

16:08

code review could be very effective to detect those

16:14

kind of issues we want to look for the violations of coding standards we want

16:26

to check to make sure coding standards are followed by the code we

16:36

also want to identify opportunities for code reuse if we find some code

16:47

components have been developed elsewhere we could make suggestions to reuse the

16:55

code instead of having similar code in different places we want to check for

17:05

robustness making sure the code provides adequate error handling in many cases the

17:13

developer focuses on the features the positive scenarios because they want to get

17:22

it up running so as a result they may not pay adequate attention on

17:31

exceptional cases on error handling readability is a very important concern

17:54

during code review we want to make sure the code is readable because otherwise

18:03

the code cannot be maintained we want to make sure the names are meaningful the

18:13

name of a variable should indicate what the variable represents the name of a

18:21

method should indicate what the method is supposed to do

18:28

we wanna make sure the code is well structured we also look for

18:40

opportunities for refactoring we will discuss refactoring in more detail after

18:51

this topic we want to check if unit tests are included and if those tests

19:05

achieve sufficient test coverage we want to make sure adequate comments are

19:17

provided in the code this is especially important for components that are more

19:28

difficult to understand next we discuss some tips we could use in practice in

19:38

general for each meeting we could plan to review between

19:45

200 and 400 lines of code that is not comments each meeting should not exceed one hour

20:00

the reason is that people just get tired after one hour the inspection rate is

20:09

about 300 lines of code each hour we could expect to find about 15 defects each hour

20:25

the number of participants could be between

20:32

three and seven we do not want to have too few people because we

20:41

want to include different perspectives each participant brings a different

20:50

perspective we do not want to have too many people

20:55

because the more people we have the more difficult to manage the process some

21:04

tips for the management code review should not be optional as we discussed before if

21:18

people know their work is going to be reviewed by other people they could do a

21:26

better job but code review could be selective this means

21:34

we do not have to review every piece of code for example we could do

21:43

code review only for code that is critical that is used by

21:51

many components of the system or code that is more complex that is more difficult

22:00

to get it right or code that is written by people who are new to the project

22:15

typically those code needs to be reviewed we could do

22:27

separate reviews for different aspects we could do a separate review

22:37

for security a separate review for memory management a separate review for

22:47

performance tips for the reviewers always keep in mind we try to critique the

23:08

code not the person sometimes it helps to ask questions when we

23:21

try to make a comment instead of trying to just make a very strong statement we

23:31

want to appreciate good things that we find in the code not only pointing out

23:39

potential issues in the code we want to remember that there are different ways

23:48

to solve a problem it may not be the way you would use it does not mean it is not

24:01

correct we want to be respectful and we want to

24:10

provide constructive comments for example you may want to give specific

24:20

ideas or actions the developer could take to

24:25

improve the quality of code some tips for the

24:34

developers remember it is the code that is being reviewed it is not you being

24:48

reviewed you want to maintain coding standards so it is typically a good idea

25:00

to check if there is any violation of coding standards in your code before you

25:07

submit the code for review you want to create a check list of the things that

25:18

code reviews typically focus on so you want to fix those things before the reviewers

25:34

find them you want to be respectful to the reviewers and you want

25:42

to be receptive to the comments and suggestions made during the review meeting

25:55

some things we don't want to do during code review

26:01

we do not want to use code review as performance measurement the reason is that

26:10

people could otherwise become very defensive we want to avoid emotions or personal

26:19

attacks and we do not want to be too defensive remember it is the code

26:28

that is being reviewed not the developer we want to avoid

26:38

ego and politics during code review very important after we distribute the review

26:48

copy we should make no additional changes

26:52

these are some reasons code review may not be effective the participants do not

27:13

understand the the review process they do not know what they are supposed

27:22

to do the reviewers try to critique the developer instead of the code that is

27:34

written by the developer reviews are not well planned the reviewers

27:43

are not well prepared remember code review is a technical

27:49

activity that requires deep thinking if people are not prepared we cannot

28:00

make meaningful comments and suggestions and then they could waste the time and

28:10

effort of other people we do not want to spend too much time on trying to address

28:21

each comment or suggestion because review meetings are not about problem solving

28:33

instead the developers could take time to address the comments and suggestions

28:41

after the review meeting the wrong people participate again code review

28:50

is a technical matter we do not want to to invite non technical people to the

28:59

review meeting reviewers focus on style not substance so people have

29:07

different preferences what is more important is the substance the content

29:16

of the solution instead of the style of the solution many tools have been

29:30

developed to support code review they can be classified into two types of tools

29:39

the first type tries to automate and manage the workflow for example these

29:51

tools could help the distribution of the review copy could help to make

29:59

comments help to manage the status of each comment the second type tries to automate

30:11

the actual inspection task this tool CheckStyle can automatically check

30:21

whether coding standards have been followed this tool can automatically check C

30:30

programs for security vulnerabilities this is a model checker for C programs

30:44

it can be used to specify and check general properties to recap what we have

31:03

discussed code review is considered to be one of the most effective ways to

31:11

increase code quality when we do code review it is important to remember it

31:21

is the code that is being reviewed not the developer code review is a good

31:30

opportunity to spread knowledge among team members it is also a good

31:40

opportunity to train new people code review is a important part of the

31:58

development process again just having code review in the development process

32:07

could help increase the quality of the code this is just because people would do a

32:16

better job when they know their work is going to be reviewed by other people

# CSE 4321 Refactoring (Part 1, Fall 2020)

00:02

in this video we discuss a new topic software refactoring first we give an

00:10

introduction to this topic what is software refactoring why we want to do

00:17

refactoring then we look at a motivating example to get a general idea about how

00:26

refactoring works we discuss some bad smells we can detect in the code

00:35

they basically indicate opportunities for refactoring what is refactoring

00:47

basically refactoring is about restructuring the source code it moves

00:55

code around to make the code easier to understand easier to modify

01:05

refactoring does not change the behavior of a program that is visible to the

01:13

outside

01:21

so from the user perspective from the outside perspective the program

01:28

has not been changed why we want to do refactoring

01:35

first refactoring can help to improve the design of a program

01:41

the design of a program could be very good in the beginning but after we make

01:48

changes to the program the design could get messy

01:54

refactoring can be used to restore the structure of a program

02:00

refactoring can be used to make software easier to understand when we write code in

02:08

many cases we first focus on how to make it work that basically means we try

02:17

to make the code understandable to the computer refactoring helps us to think

02:26

about how to make it understandable to the people code being easier to

02:35

understand is very important for maintenance because if we cannot

02:41

understand the code we cannot maintain the code refactoring can help to find

02:50

faults that may exist in the code when we do refactoring it makes

02:56

us to think deep about the program and doing that in many cases can help to

03:05

detect bugs that may exist in the code refactoring could help to write

03:14

code faster this is because refactoring can improve the design of a program

03:22

it can improve the structure of a program

03:26

a good design a good structure can help to make changes easier when

03:39

we want to do refactoring when we add a function sometimes we may say to

03:47

ourselves if only I have designed the code in a

03:53

different way adding this function would be much easier

03:59

this is an indication that we want to refactor the code before we add the

04:06

function and sometimes when we try to fix a bug it also indicates a opportunity

04:17

for refactoring this is because the very existence of a bug indicates that

04:25

the code is not well-structured so we may want to refactor the code

04:33

before or after we fix the bug when we do code review we get comments

04:41

suggestions about how to restructure the code so that is also a

04:49

good time to do refactoring some potential problems with refactoring when we do

05:05

refactoring we move code around in some cases this could change the

05:13

interfaces whenever we change the interface we want to be careful this

05:22

is because some other components may depend on the interface if we change the

05:30

interface we are at risk to break the rest of the system one approach

05:39

to address this problem is to keep the old interface until the users had a

05:48

chance to switch to the new interface in some cases the code may be designed in a

05:58

way that is very difficult to change the

06:06

implication of this problem is that when we make a design decision we want to ask

06:14

how difficult it would be if later we have to change the design decision

06:24

if it is easy to change a design decision then you do not need to

06:31

worry too much about the decision if the decision is very difficult to change it

06:40

would be a good idea for you to be more careful spend more time on the

06:47

decision you may want to talk to other people you may want to discuss with other

06:57

people before you make the decision

07:10

so when not to refactor if it is easier to

07:15

write from scratch if the current code just doesn't work you may not want to refactor

07:37

also when it is very close to a project

07:41

deadline you may not want to refactor the reason is that the long term benefit of

07:50

refactoring may not appear until after the deadline

# CSE 4321 Refactoring (Part 2, Fall 2020)

00:04

next we look at a motivating example to get some general idea about how to do

00:11

refactoring this is a video rental system it manages video rentals we have

00:25

three types of movies children's movie regular movie

00:29

new release movie each movie has a title and has a price code depending

00:38

on the type of the movie we have a constructor then we have this method to

00:47

get the price code this method is used to set the price code this method is to

00:57

get the title of the movie this class represents a movie rental so in this

01:14

class we have the movie object the movie that is being rented and the days

01:24

the movie is rented for we have a constructor we have this

01:37

method to get the number of days rented and this method to get the

01:44

movie object associated with this rental object then we have this class that

01:59

represents a customer this is the name of the customer this vector represents

02:09

all the rentals the customer has we have a constructor and we could add a

02:19

rental to the collection of rentals this customer has we could get the

02:27

name of the customer then we have this method that prints out the statement for

02:35

the customer this is the main method we're going to focus on this is the

03:03

implementation of the statement method so in this method we have a loop so this

03:14

loop basically goes through every rental this customer has for each

03:22

rental it determines the amount the customer has to pay depending on

03:33

the type of the movie and the number of days the movie is rented then it

03:42

computes the frequent renter points then it prints out the information for each

03:53

rental including the title of the movie being rented the amount has to

04:01

pay for this rental and out side the loop we print out the total amount

04:13

the customer has to pay and the frequent renter

04:19

points the customer has earned from the rentals the

04:25

customer has made so basically this

04:32

system has three classes movie rental and customer and the main computation is

04:51

inside the statement method the other methods are simple methods to access the

05:03

data members of each class so what is your opinion about the design of this program

05:15

is this a good design first the program does not seem to be balanced if

05:24

we look at the code the customer class

05:35

is much bigger than the other two

05:39

classes and the statement method has almost all the business logic in

05:52

addition consider what happens if we have to add a new method to print the

06:00

statement in a different format in particular could we reuse this existing

06:16

statement method if we want to print the statement in a different format

06:31

in

06:32

the next slides we're going to see how we could restructure

06:38

the code to improve the design the first step when we do refactoring is to

06:50

build a good test set this is important because we could make significant

06:57

changes during refactoring we could use this test set to make sure the system is not

07:06

broken after we finish refactoring for this

07:11

program we could create some customers each customer could rent a couple of movies

07:21

of

07:22

different types and then generate statements

07:28

the first refactoring we do

07:30

is to separate computation from presentation so that we could reuse the

07:38

same computation in the new method to print the statement in the HTML format

07:45

if we look at the existing statement method

07:56

this code is computation because it basically computes the amount the customer has to

08:04

pay for each rental this is also computation it computes the frequent

08:13

renter points this computes the total amount

08:23

so first we extract this computation out of the statement method if

08:42

we look at this code it uses information in the rental class so we move the code out

08:51

of the statement method and define a method in the rental class

09:11

this is the new method in the rental class in the statement method

09:30

we just call this getCharge method

09:35

to get the amount the customer has to pay for each rental we also call this method to

09:45

compute the total amount the customer has to pay then we do the same for the

09:57

computation to get frequent renter points so basically we define a new

10:05

method in the rental class to get the frequent renter points then in the

10:16

statement method we just call this method

10:35

next we extract the computation

10:39

for total amount out of the statement method as well so if you look at this method

10:56

basically we iterate through all the rentals for

11:03

each rental we get the amount we

11:08

have to pay for each rental and then we add it up to get the total amount we

11:18

have to pay for all the rentals

11:38

and we do the same for the total frequent

11:44

renter points so basically we iterate through all the rentals and we get the

11:56

frequent rental points for each rental and then we add it up to get the total

12:08

frequent renter points after we extract those computations out we can have the

12:27

new statement method so in the new

12:39

statement method we now can reuse the code

12:45

we can reuse the computations we have extracted out if we look at the code

12:53

we call this method to get the amount we have to pay for each rental and we

13:03

call this method to get the total amount

13:12

call this method to get the total frequent renter

13:17

points now if you look at this statement method and the existing

13:34

statement method they have different presentations but they use the same

13:44

computation there is a general guideline about the use of a switch statement so

14:00

basically the idea is that if we use a switch statement in a class A then we

14:10

want to switch on the data members defined in class A we do not want to switch on a data

14:20

member defined in a different class say B the reason is that if the data members

14:28

of class B change we may forget to change the switch statement in class A this is

14:40

particularly so in a team development environment because class A and class B may

14:51

be

14:52

developed by different people so when we make changes in Class B we may not know

15:04

that we have a switch statement in a different class A that

15:09

depends on these data members let us look at this method recall that this method

15:21

computes the amount we have to pay for each rental if you look at this switch

15:31

statement so basically it checks the price code of the movie but this price code

15:43

is defined in the movie class not in the rental class so we want to move this

15:52

method from the rental class to the movie class and this is what we have

16:03

imagine if

16:04

we decide to add a new movie type then we have

16:08

to change this switch statement we have

16:14

to add a new case because now this method is defined in this movie class

16:23

when we add the new movie type it is easy for us to see that this switch statement

16:33

also needs to be changed we need to add a new case branch in this method and we

16:45

can do the same for get frequent renter points we want to move this method

16:55

from the rental class to the movie class because here we check the price code

17:07

that is defined in the movie class

17:21

we move that method to the movie class and we call

17:29

this method in the rental class let us revisit

18:04

this method so this method basically computes the amount we have to pay for each

18:13

rental in this method we have this switch statement this is basically a

18:21

conditional statement it checks the movie type for each movie type we do

18:31

some different computation to get the result consider what happens if we have

18:43

a new movie type let's say we have a new rental like game then we have to change this

18:55

switch statement we have to add a new case branch in this conditional

19:04

statement a different way to compute the charge is that we could take

19:13

advantage of inheritance a signature feature in object-oriented programming so

19:21

basically we have a base class called Movie then we have three subclasses

19:35

each subclass represents a different type of movie in each subclass we compute

19:45

the

19:46

charge for this particular type of movie so using this inheritance-based design we get

19:58

rid of the switch statement if we have to add a new type we just need to add a new subclass

20:08

in this inheritance hierarchy we do not have to change the code for the other

20:17

existing movie types can we do even better let us consider what changes we have to

20:37

make if a new release movie becomes a regular movie

20:44

after some time ideally we would like to reuse the new

20:56

release movie object as much as possible so we would like to directly change a new

21:08

release movie object to a regular movie object

21:14

this is however not possible because these are two different types and the regular

21:24

movie is not a parent type of the New Release movie so we cannot

21:33

directly change the type of a new release movie

21:38

object to regular movie instead what we have

21:43

to do is that we have to create a new regular movie object we have to copy

21:52

the information in the new release movie object to the new regular movie

22:01

object except that this new movie object has a different movie type which is

22:09

Movie\_REGULAR that indicates this new movie object is a regular movie object this

22:20

is

22:21

a different design in this design each movie has a price object and

22:31

we have the base class called Price class then

22:37

we have three different subclasses each

22:41

subclass represents a different movie type so a new

22:48

movie would have a new release price object a

22:59

a regular movie object would have a regular price

23:04

object in this design if we want to

23:11

change a new release movie object to a regular movie

23:15

object we only need to change the price

23:19

object of the movie object from a new release price object

23:25

to a regular price object

23:29

doing that allows us to reuse the rest of the movie object we do not

23:36

have to create an entirely new movie object

# CSE 4321 Refactoring (Part 3, Fall 2020)

00:01

next we discuss some bad smells we could detect

00:06

in the source code

00:08

these bad smells indicate opportunities for refactoring

00:17

duplicate code so basically we have the same code

00:21

structure in different places

00:24

this makes the program unnecessarily long and more importantly duplicates

00:33

make it difficult to maintain consistency in general we want to be

00:41

sensitive about duplicates the reason is that if we change one of the

00:48

duplicates we have to make sure we change all the duplicates and it can be very

00:57

easy for us to forget to do that to remove duplicate code we could extract the

01:12

common code out and then call the common code from different places long method

01:25

so

01:26

basically we have a method that has too many statements the longer a method

01:32

the more difficult it is to understand and maintain the method in addition the

01:39

longer a method the more difficult to reuse the method to remove long methods

01:48

we could divide a long method into multiple smaller methods this is similar to

02:06

long method so basically we have a very big class the class could have too many data

02:14

members or too much code in the class it is difficult to understand and

02:22

maintain a big class similarly we could break a big class into

02:29

smaller ones based on the relationship between the different parts of the big

02:36

class in this case a method takes too many

02:46

parameters in the signature when a method

02:50

takes too many parameters the method can be difficult to understand and use

03:00

in addition the more parameters we have the less stable the interface is the

03:09

reason is that if we make a change to any of the parameters the interface is

03:17

also changed in general we want to be very sensitive about interface changes because

03:25

it could break the existing system

03:31

to deal with this situation we could use an object to contain the data so basically

03:39

we have one object parameter inside this object parameter we could have multiple

03:48

data

03:49

members that represent the original individual parameters in this case a

03:58

method seems to be more interested in a class different from the class

04:05

this method is defined so basically if you look at the code of the method it

04:16

uses more data members defined in a different class the potential problem

04:23

is that the method may not be in sync with the data it operates on the

04:32

reason is that the data is in a different class if we make a change in that class

04:39

we may forget to change this method

04:43

this is particularly so if the method is defined in a class that is developed by

04:51

a different person than the class in which the data is defined to deal with this

05:05

situation we could move the method to the class that defines the most data

05:14

needed by the method in this case the code provides excessive support for

05:28

generality that is not really needed so what happens is that a good design

05:39

should anticipate changes that could happen in

05:44

the future but this does not mean the

05:48

code should be designed in a way that can support any possible changes in the

05:56

future instead we want to look at the possible changes to see which ones are

06:05

more likely to happen so we could design the code in a way that

06:16

supports the changes that are likely to happen the problem with excessive

06:25

support is that it makes the code more

06:29

complex than necessary and the more complex the code the more difficult to

06:37

maintain the code to deal with this situation we could remove the additional

06:46

support that is not needed or the support

06:49

that is designed for changes that are not likely to happen data classes

07:09

are basically data holders they only have data members and get set methods for

07:18

those data members they do not have significant computation logic so those

07:27

classes are typically accessed by other classes and that means there is a very

07:37

high degree of dependencies between those classes to deal with this

07:46

situation we want to assign more responsibilities to these classes

07:54

if you consider the motivating example in the original version the movie and rental

08:02

classes are data classes we gave them more responsibilities to compute the amount to

08:12

be charged and the frequent renter points during refactoring

08:22

in this

08:23

case we have too many comments this can be a signal that the code is difficult

08:32

to understand and that is the reason why

08:36

the comments are provided for the code so we could try to remove the comments

08:46

by refactoring the idea is that maybe if we could restructure

08:52

the code in a different way then those comments may not be needed so basically

09:01

if we see too many comments for a component we want to ask the question

09:08

are those comments really needed could we change the structure of the code so

09:15

that we do not need this many comments

# CSE 4321 Refactoring (Part 4, Fall 2020)

00:01

next we recap what we have discussed

00:08

refactoring is basically moving code around to improve the code structure so

00:15

that it is easier to make changes in the future very important refactoring does

00:22

not change the external behavior of a system so from the user perspective we

00:30

still have the same system refactoring allows us to do a reasonable design

00:39

it does not have to be perfect the reason is that we could always come back

00:47

later to refactor the code to improve the design in general computation should be

00:56

separate from presentation this is particularly important these days

01:05

because we have so many different platforms we have smartphones we have

01:13

iPads we have desktops when we develop an app we want the app to be used in

01:22

different platforms we discussed different bad smells that we could

01:30

detect in the source code they indicate opportunities for refactoring