The Rocky Horror Code Show

Why refactoring is not an option, but a necessity

Thomas Kashofer
Director Expert

2023-07-13

CGI

Agenda

01 Motivation

02 Technical debt

03 Funny WTFs

04 Results

05 Reasons

06 Recommendation

Thomas Kashofer

Thomas Kashofer

- 22 years of professional experience
- 10 years of personnel responsibility
- Various roles (developer, tester, requirements engineer, technical architect, solution architect, project manager, consultant)
- Software archaeologist
- Open-source enthusiast

Director Consulting Expert
Digital Transformation Consultant
thomas.kashofer@cgi.com
+49 151/16358528
linkedin.com/in/thomas-kashofer-52468625
xing.com/profile/Thomas_Kashofer

"To create an environment in which we enjoy working together and, as owners, contribute to building a company we can be proud of."

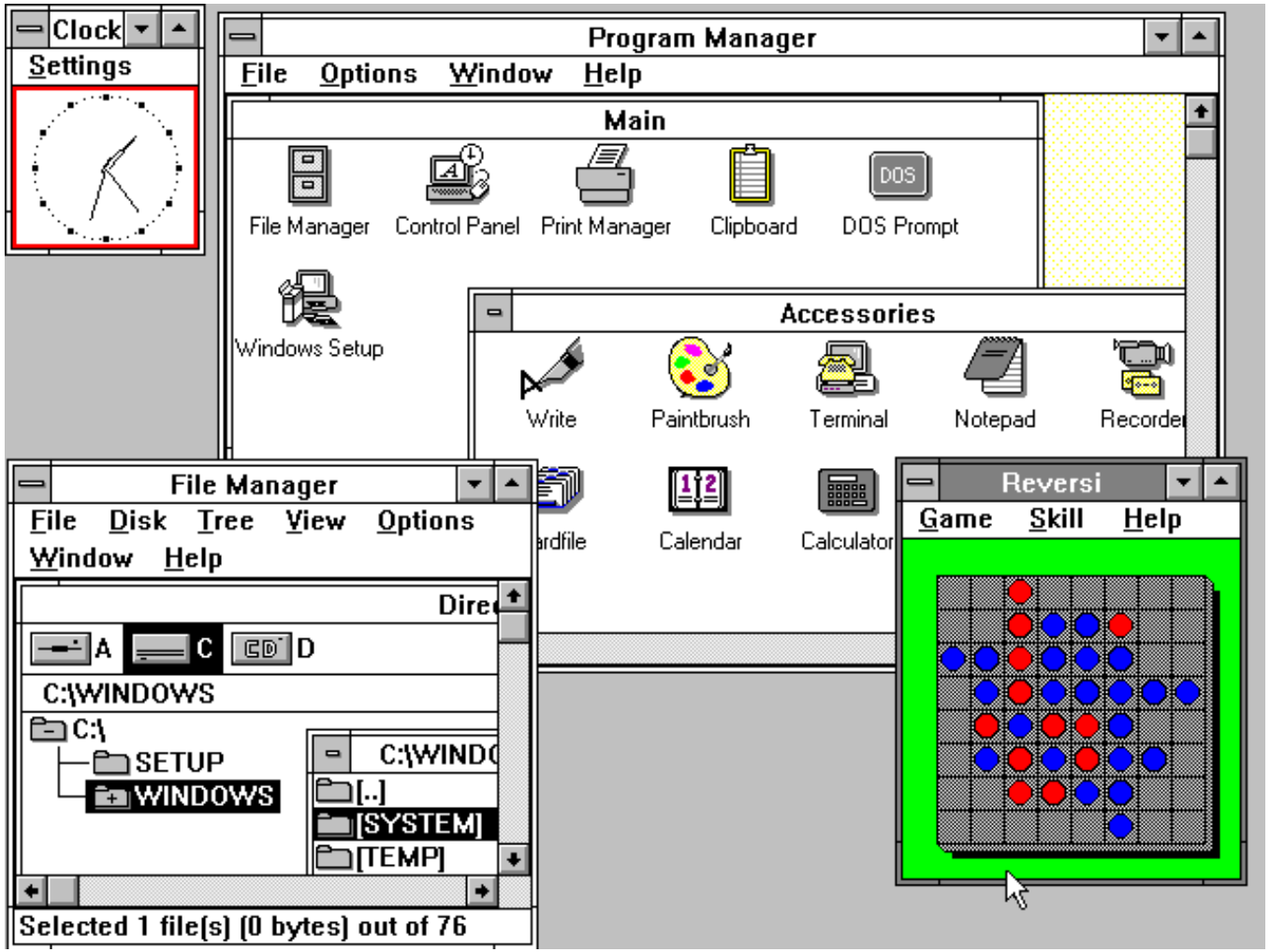
Thomas Kashofer

1984 – my 1st Computer

Source: https://zock.com

Thomas Kashofer

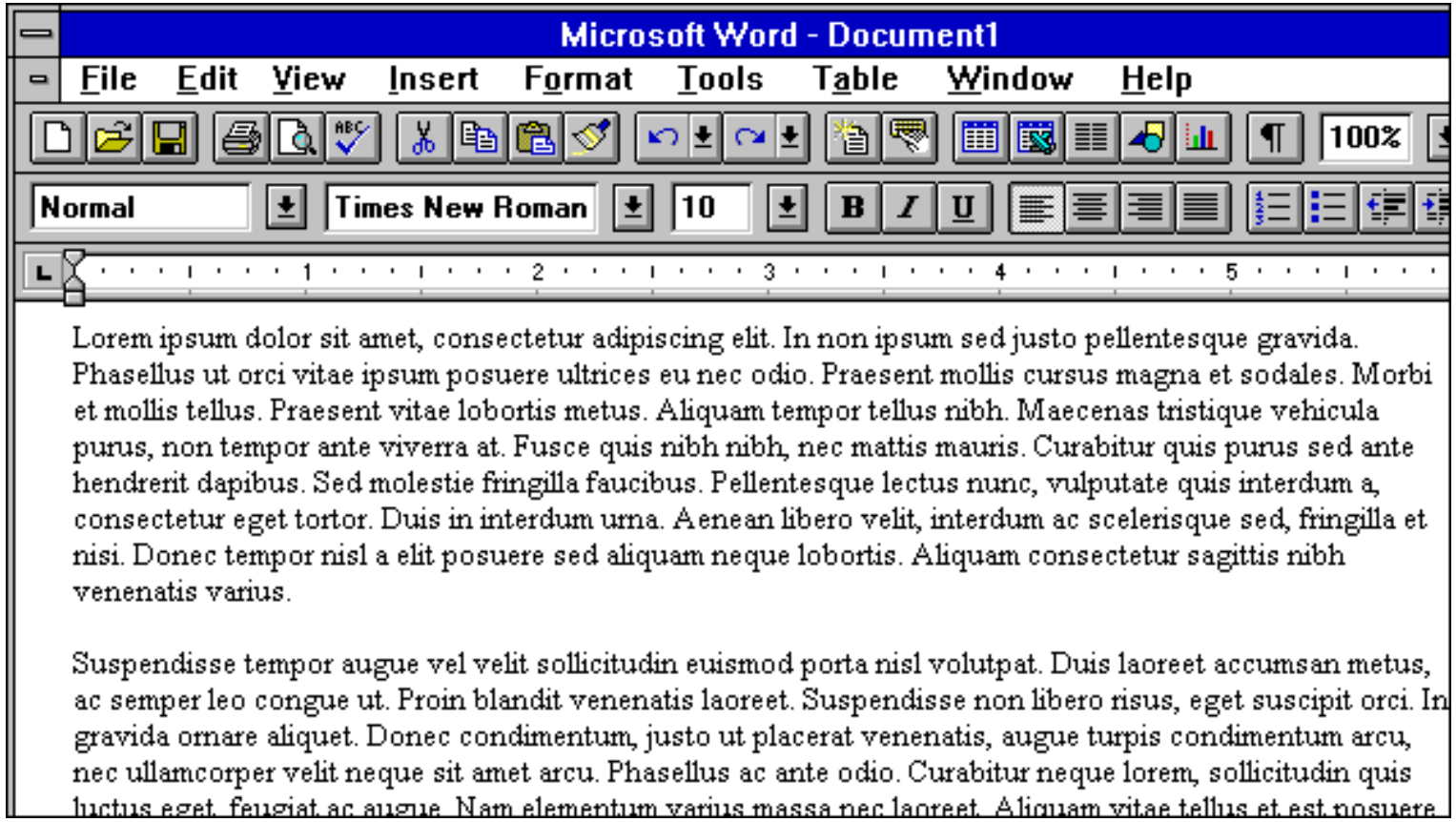
1990 – my 1st Windows – 3.0



Source: https://en.wikipedia.org/wiki/Windows_3.0#/media/File:Windows_3.0_workspace.png

Thomas Kashofer

1993 – my 1st Word for Windows – 6.0



Source: https://www.techjunkie.com/retro-friday-microsoft-word-6-0/

Thomas Kashofer

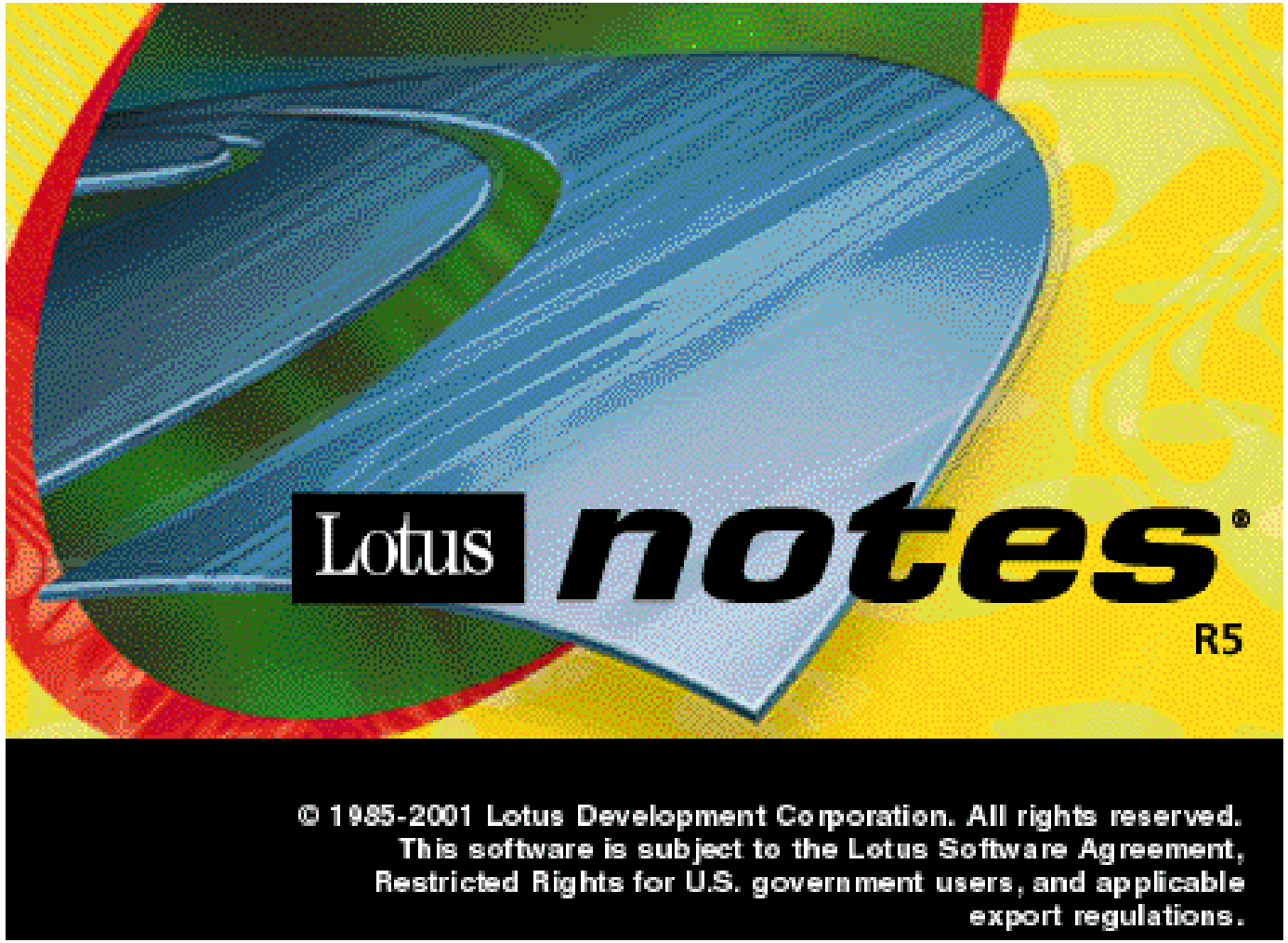
1993 – my 1st Linux – self-compiled

```
phoenixnap@test-system:~$ ll
total 64
-rw-rw-r-- 1 phoenixnap phoenixnap  106 Jul 21 14:34 check_root.sh
drwxrwxr-x 5 phoenixnap phoenixnap 4096 Apr 15 11:33 chroot_jail
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Desktop
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Documents
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Downloads
-rw-rw-r-- 1 phoenixnap phoenixnap  110 Jul 21 13:11 example_bash.sh
-rw-rw-r-- 1 phoenixnap phoenixnap   27 Jul 21 13:09 example_config.sh
-rw-rw-r-- 1 phoenixnap phoenixnap   83 Jul 21 14:33 example_script.sh
-rw-rw-r-- 1 phoenixnap phoenixnap   14 Jul 22 12:52 example.txt
-rw-rw-r-- 1 phoenixnap phoenixnap   65 Jul 21 14:32 foo.sh
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Music
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Pictures
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Public
drwxrwxr-x 3 phoenixnap phoenixnap 4096 Jul 21 12:58 source_command
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Templates
drwxr-xr-x 2 phoenixnap phoenixnap 4096 Apr 15 11:25 Videos
phoenixnap@test-system:~$
```

Source: https://phoenixnap.com/kb/linux-source-command

Thomas Kashofer

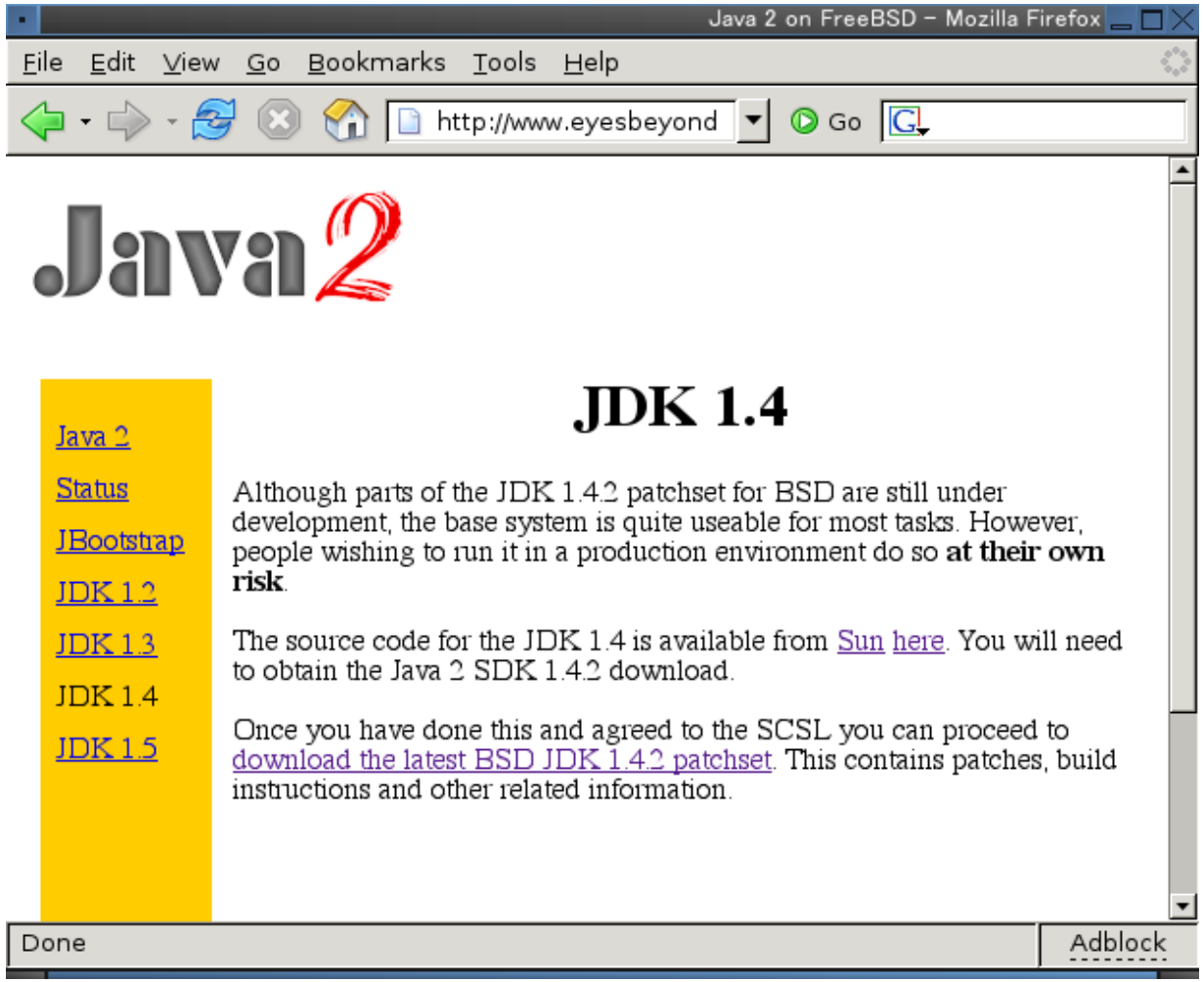
2001 – my 1st Job as Lotus Notes/Domino Developer



Source: https://notesapplicationmigration.com/lotus-com-notes-domino-wikis-and-forums/

Thomas Kashofer

2002 – my 1st Java v1.4



Source: https://www.ocf.berkeley.edu/~reinholz/freebsd/jdk14.html

Agenda

01 Motivation

02 Technical debt

03 Funny WTFs

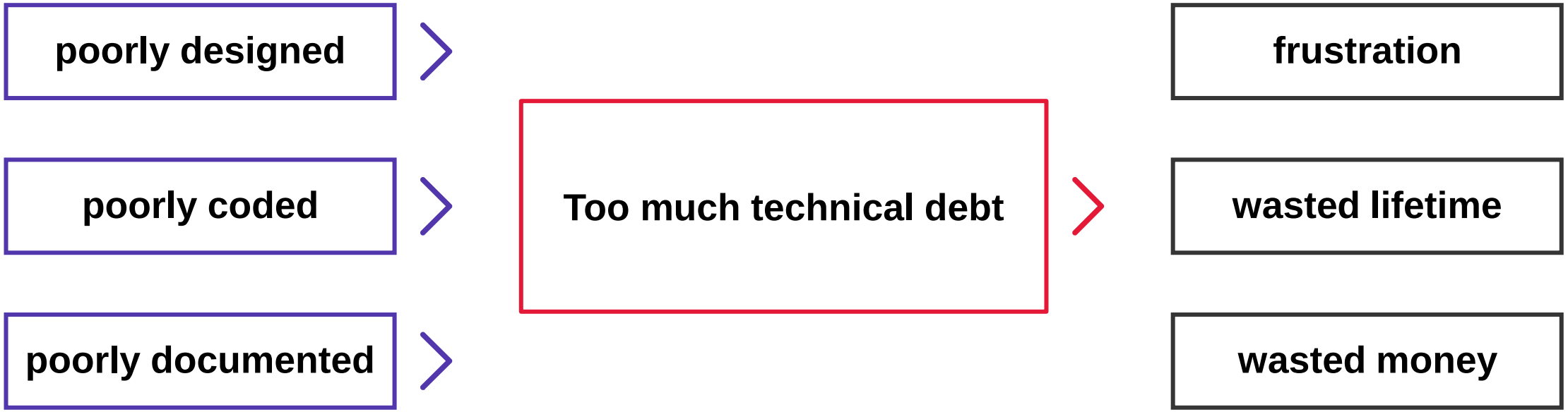
04 Results

05 Reasons

06 Recommendation

My motivation

Many of the systems I have reviewed in the last years have quite a few things in common:



Agenda

01 Motivation

02 Technical debt

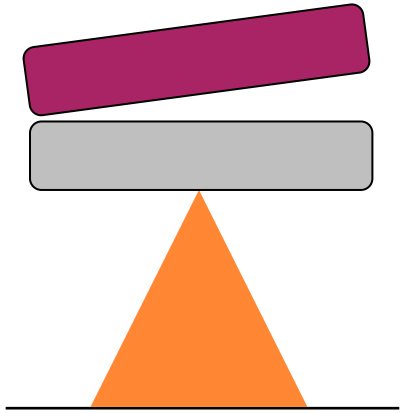
03 Funny WTFs

04 Results

05 Reasons

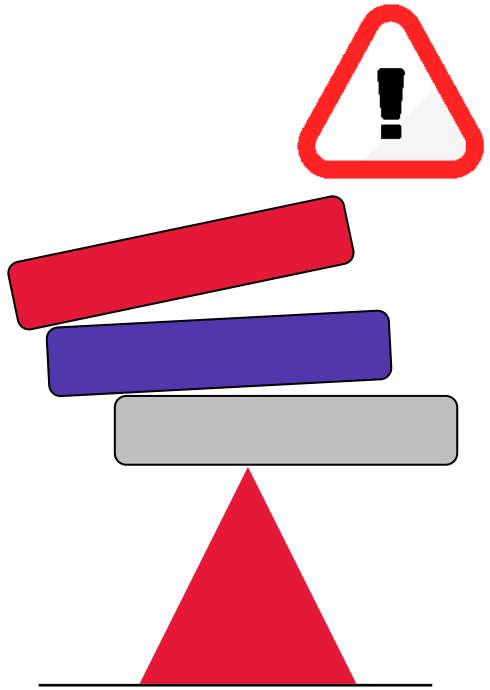
06 Recommendation

Technical debt – When speed is the only aim



Intentional

Occurs when an organization makes a conscious decision to optimize for the present rathen than for the future.



Unintentional

Occurs when an organization makes an avoidable mistake.

The 4 Reasons for Technical Debt

	Intentional	Unintentional
Reckless	We don’t have time	We do not know how
Prudent	We will deal with it later	We should not have done that

Technical debt - Examples

Common technical debts in software development projects:

- no code comments / lots of comments
- meaningless or misleading names (variables, methods, …)
- long methods
- methods that does many things
- missing or patchy documentation
- missing or patchy tests
- missing CI/CD infrastructure

Technical debt - Examples

More technical debts in software development projects:

- missing logging framework/concept
- use of coding anti-pattern
- missing coding standards, incl. development and deployment workflow
- disregarding of compiler warnings and static code analysis results
- disregarding of TODO- / FIXME- / XXX-comments in the code

Agenda

01 Motivation

02 Technical debt

03 Funny WTFs

04 Results

05 Reasons

06 Recommendation

WTFs

WTF = Worse Than Failure

(see https://en.wikipedia.org/wiki/The_Daily_WTF)

WTF = Work that Frustrates

WTF = code that consists of Workarounds, ToDos and Fixes

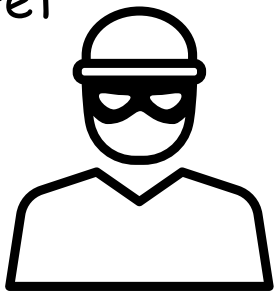
Security

Why is the “need to know” principle key?
I want to make sure, that my secrets stay secret.

Security: Dealing with Sensitive Information

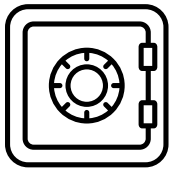
Bad:
Configuration files with sensitive Information in the Repository

```
spring.datasource.url=jdbc:postgresql://
mydatabase
spring.datasource.username=a8097378e
spring.datasource.password=secret
```



Can be misused by anyone with READ access to the repo (code scanners) or to build artifacts (admins).

Good:
Use vaults for credentials & config mgmt. system



```
withCredentials([
        usernamePassword(credentialsId:
'myApp-blackduck-token-myUser',
usernameVariable: 'USERNAME_BLACKDUCK',
passwordVariable: 'PASSWORD_BLACKDUCK')
....
scan --token=${PASSWORD_BLACKDUCK}
```

What you do not know, you cannot misuse.

Security: Dealing with Sensitive Information

Bad:
Logging sensitive information
(on INFO level)

```
log.info("jwt JWT token: {} 84!, jwt);
```

Can be misused by anyone with READ access to log files (admins, serviceDesk) and are stored in log archives.

Good:
Such information belong (if at all) into DEBUG level etc.

```
log.debug
```

and do not set the default log-level to DEBUG

What you do not log, nobody can misuse.

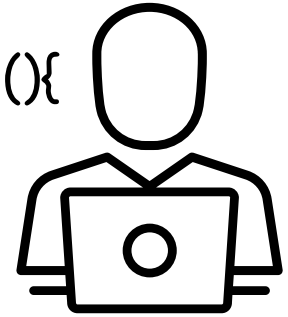
Testing

Why is code quality also important for unit tests?
I want to make sure, that I test all my code.

Testing: Easy maintenance….NOT

Bad:
Using non descriptive test case names or just numbering them

```
@test
public void getMasterDataCase1(){
…
};
public void getMasterDataCase2(){
…
};
```



Class name is not enough to know the content and intention of the test.

Good:
Use a very precise (and short) name

[UnitOfWork_StateUnderTest_ExpectedBehavior]

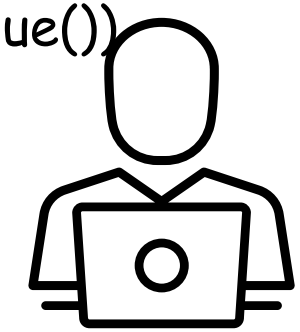
```
@test
public void
Invoice_WhenQuantityIsMissing_CannotBePro
cessed{
…
};
```

All relevant information available - saves time!

Testing: I got everything covered….NOT

Bad:
Only testing for the OK or ERROR response code but not for any values

```
@test
public void getMasterDataLangEN(){
…
assertEquals(HttpStatus.OK.value())
…
};
```



You do not know if the value itself is correct, thus you may miss relation errors.

Good:
In addition(!) testing with test data

```
@test
public void getMasterDataLangEN(){
…
assertTrue(expectedList.contains((actualEntry
)));
…
};
```

You are sure that you not only get A result, but the CORRECT one.

Maintainability

Do I really want to maintain that code later on?
I want to make it simple for myself.

Maintainability: Variable names

Bad:
Unreadable variable names

```
export class TireServiceBean {
    agts_cmplt_fltng ? : number;
    agts_ftng_rn_flt ? : number;
    agts_blcng ? : number;
    agts_whl_str_grnd ? : number;
    agts_agmt_id ? : number;
    agts_fcm_id ? : number;
};
```

If you have to guess you could be wrong and it costs time.

Good:
Variable names that so precise, that everyone understands them immediately

It saves time and eliminates the need for (much) additional documentation.

Maintainability: Variable names

Bad:
Unreadable variable names

```
@Column(name = „aspir_rec_id")
@Column(name = „aspir_cntry_cd")
@Column(name = „aspir_dlr_cd")
@Column(name = „aspir_01")
@Column(name = „aspir_02")
@Column(name = „aspir_rec_typ")
@Column(name = „aspir_03")
```

If you have to guess you could be wrong and it costs time.

Good:
Variable names that so precise, that everyone understands them immediately

It saves time and eliminates the need for (much) additional documentation.

Maintainability: Magic numbers

Bad:
Magic numbers

```
openCoockieStatement(value:any){ class
    this.booleanFlag=value;
    if(value==2)
    {
      this.CookieFooter=false;
    }
    this.cookieStatement=true;
};
```

It costs time to check what is behind those numbers.

Good:
Meaningful names that so precise, that everyone understands them immediately or a clear inline documentation about the hidden meaning

It saves time and eliminates the need for (much) additional documentation.

Maintainability: No obvious misleadings

Bad:
When types and names to not match the content and/or the intended usage

```
someClass(){ class
    string myBoolean;
    if(myBoolean==3)
    {
       do something;
    }
    do somethingElse;
};
```

It costs time to check what is behind those numbers.

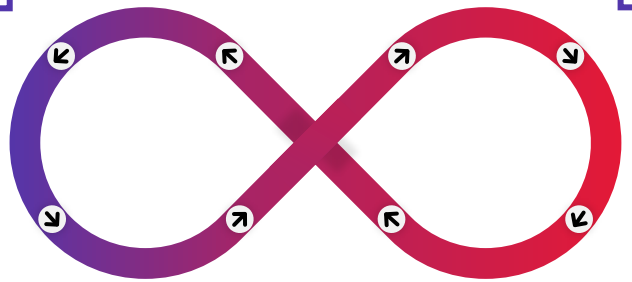
Good:
The variable types should match their values and the intended usage.

It saves time and eliminates the chance of mentally running into the wrong direction.

Maintainability: Code complexity vs. Code duplication

Higher code complexity

More code duplication



Less code duplication

Less code complexity

Maintainability: Repository sanity

Bad:
Duplicated trees checked in after renaming the top directory

```
repo/
├── daihatsu
│   ├── dir-1
│   │   ├── file-1
│   │   ├── file-2
│   │   ├── file-3
│   │   ├── file-4
│   │   └── file-5
│   ├── dir-2
│   │   ├── file-1
│   │   ├── file-2
│   │   ├── file-3
│   │   ├── file-4
│   │   └── file-5
│   ├── file-1
│   ├── file-2
│   ├── file-3
│   ├── file-4
│   └── file-5
└── diahatsu
    ├── dir-1
    │   ├── file-1
    │   ├── file-2
    │   ├── file-3
    │   ├── file-4
    │   └── file-5
    ├── file-1
    ├── file-2
    ├── file-3
    ├── file-4
    └── file-5
```

Good:
Delete the old code tree, it will remain in the repository anyway

If you have to guess you could be wrong and it costs time.

It saves time and eliminates the chance of changing irrelevant code.

Maintainability: commented out code

Bad:
Commented out complete classes or the core class code in the active master branch

Good:
Delete continuously commented out sections, since it will remain in the repository anyway

```
@Scheduled(cron = "${cron.expression}")
private void scheduledCronTask(){
  List<Object[]> custDataResponse = get();
    if(!CollectionUtil.isEmpty(custResponse)) {
      for (Object[] cust : custDataResponse) {
        // doSomething;
      }
    }
};
```

It costs time to check what code is really active.

No „mentally deleting code“ necessary and more space in the IDE for the rest of the code.

Agenda

01 Motivation

02 Technical debt

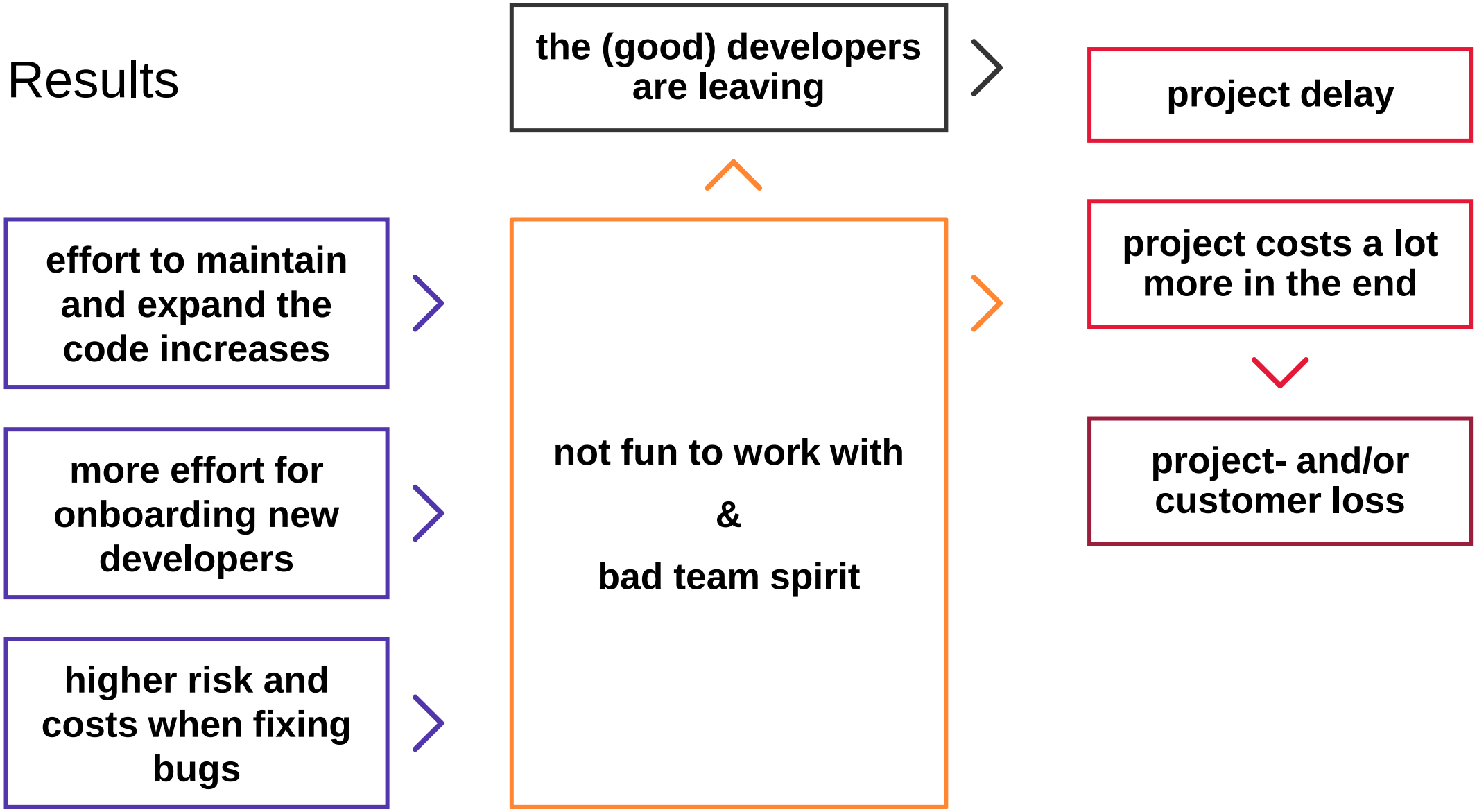
03 Funny WTFs

04 Results

05 Reasons

06 Recommendation

Results


the (good) developers are leaving
project delay
effort to maintain and expand the code increases
not fun to work with
&
bad team spirit
project costs a lot more in the end
more effort for onboarding new developers
project- and/or customer loss
higher risk and costs when fixing bugs

Results

Code that is hard to maintain is very costly !!!

Agenda

01 Motivation

02 Technical debt

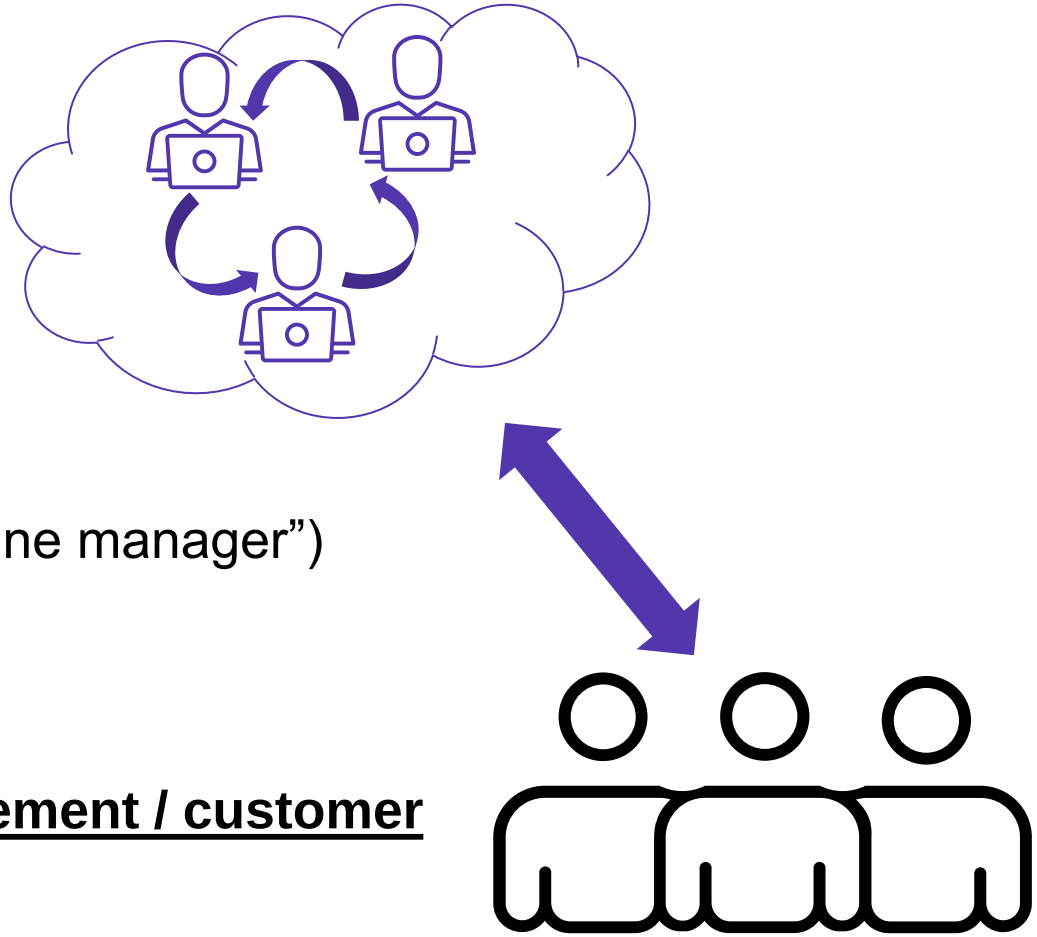
03 Funny WTFs

04 Results

05 Reasons

06 Recommendation

Reasons



Source: members of development team

Not enough skills

No plan at the beginning

Fear (“I cannot tell this to my project manager / line manager”)

<u>Source: project management / line management / customer</u>

Not enough budget

Not enough time, leading to too much time pressure (“haste makes waste”)

No reviews from (experienced) peers

Agenda

01 Motivation

02 Technical debt

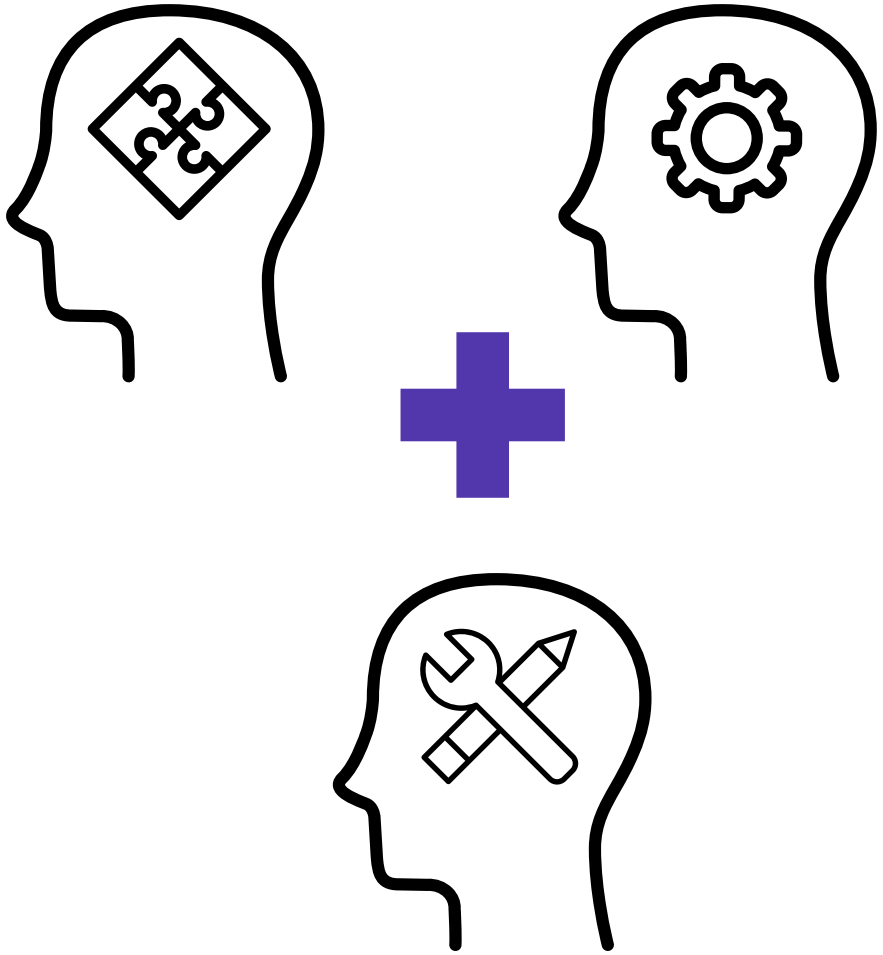
03 Funny WTFs

04 Results

05 Reasons

06 Recommendation

Less artists – more engineers and craftsmen



Recommendations

Planning

- No pure refactoring sprints, but budgeting 5-10% for it right from the beginning
- Training and peer-reviews

Process

- everyone in the development team is encouraged to make improvements within the project
- establish a change culture in the team

Communication

- - open and straight communication to customer and own management
  - communicate issues and fix them in a timely manner, instead of trying to cover them up
  - communicate the advantages of a continuous refactoring repeatedly

Costs

Assumptions

Team of 6 developers

Bad code costs each developer 16 min per day

16 x 6 = 96 min per day

96 x 5 = 480 min / 8h per week

Conclusion

Team looses 1day per week for one developer

In 6 months this project wastes 24 days (3,3%)

Suggestion

Use that 1day per week to improve the code

“Just do it”

Results

Time/Cost savings & a more motivated development team

Refactoring

NEVER BE

AFRAID

TO EXPERIMENT

Sources

Websites and blogs

https://thedailywtf.com/

https://blog.codinghorror.com/

https://muhammad-rahmatullah.medium.com/wtf-per-minute-an-actual-measurement-for-code-quality-780914bf9d4b

https://blog.devgenius.io/the-best-examples-of-bad-code-ive-come-across-production-mode-4f13e8d4de2

https://www.quora.com/What-are-some-examples-of-bad-code

https://en.wikipedia.org/wiki/Just_another_Perl_hacker

Supplementory

Skill

Reasons: Skill

Boolean usage

```
string result = "fix";
if (flag == true)
{
result = "pre" + result;
}
if (flag == false)
{
result = "post" + result;
}
return result;
```

Comparing flag against true is redundant when its a boolean flag to begin with.
Using a separate if-clause to handle the alternate condition is redundant, when it should've been an else clause.

Suggestions

```
return (flag? "prefix" : "postfix");
```

or

```
if (flag)
{
    return "prefix";
}
else
{
    return "postfix";
}
```

And most importantly, when someone uses a code-search tool to find all instances of "prefix", they won't get Zero-Results-Found like in the original code.

Reasons: Skill

Do NOT create software as an IQ test

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU / lreP rehtona tsuJ";sub p{
@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=$f=!fork;map{$P=$P[$f^ord
($p{$_})&6];$p{$_}=/ ^$P/ix?$P:close$_}keys%p}p;p;p;p;p;map{$p{$_}=~/^[P.]/&&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$_=$d[$q];sleep rand(2)if/\S/;print
```

Forking processes to print out one letter each in the correct order.
(an example in the “Just another perl hacker” (JAPH) challenge)

KISS principle – Do NOT make it more complex that it needs to be

Would you understand it at 2 a.m. in the morning?
Could you explain it to a Junior Developer in a few minutes?