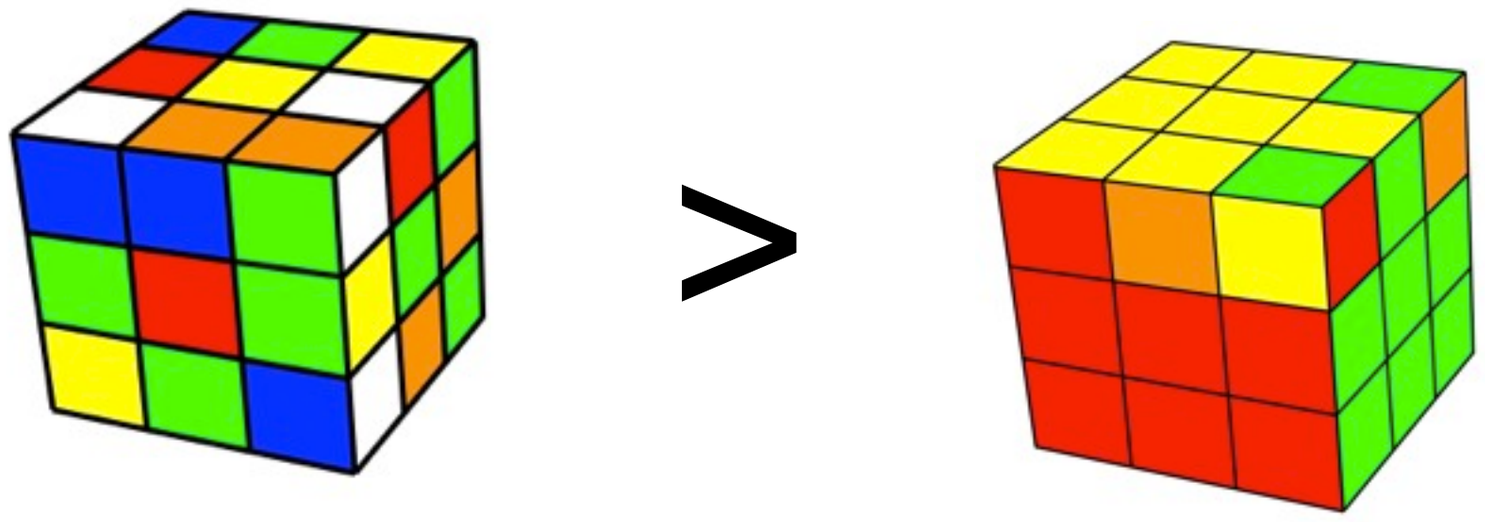
# Code Entropy and Physics of Software

Olve Maudal
@olvemaudal



Ved å studere hvordan en kodebase endrer seg over tid kan man observere hvordan svake og sterke krefter beveger koden i bestemte retninger. Det er særlig summen av de små endringene som er interessante. Vi har studert et par kjente open-source prosjekter for å identifisere noen av de rådende kreftene. I denne sesjonen vil vi først introdusere konseptet "code entropy" for så å vurdere og diskutere tilstanden til sammenlignbare kodesnutter.

En 60 minutters sesjon på JavaZone, September 7-8, 2011

(This is based on work partly done in collaboration with Jon Jagger)

(facsimile from my Smidig 2008 talk)

# (facsimile from my Smidig 2008 talk)

## Code Cleaning

**some techniques for improving existing code**

A fundamental condition for large scale agile and effective software development is that the codebase is in a healthy state and easy to work with. If you do not take care of your codebase it will rot and your project (and company) will probably fail.

Clean code is code that looks like it is written by somebody who cares, and where there is nothing obvious that you can do to make it better(*). This talk will discuss some techniques and tricks for code cleaning; it might give you an idea about how to improve existing code, how to keep your codebase healthy.

Olve Maudal
oma@pvv.org

10 minute lightening talk at Smidig 2008
October 9-10, 2008

# (facsimile from my Smidig 2008 talk)

## Code Cleaning

**some techniques for improving existing code**

A fundamental condition for large scale agile and effective software development is that the codebase is in a healthy state and easy to work with. If you do not take care of your codebase it will rot and your project (and company) will probably fail.

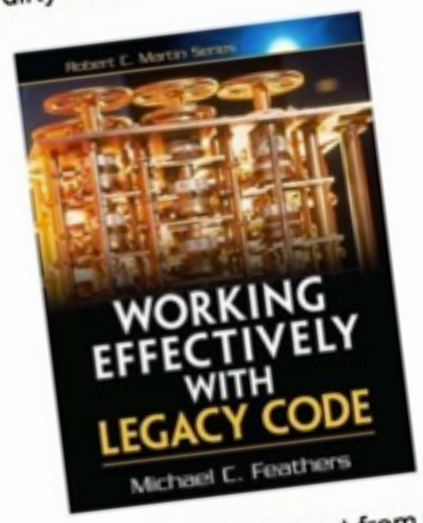
Clean code is code that looks like it is written by somebody who cares, and where there is nothing obvious that you can do to make it better(*). This talk will discuss some techniques and tricks for code cleaning; it might give you an idea about how to improve existing code, how to keep your codebase healthy.

Olve Maudal
oma@pvv.org

10 minute lightening talk at Smidig 20
October 9-10, 2008

## Background and Disclaimer

This talk is very much inspired by Uncle Bob's latest book about writing clean code, but also by Michael Feathers book about working with dirty code.


Many examples, sentences and ideas in this talk are just ripped out from these excellent books.

I agree with most of the stuff I present here...

# (facsimile from my Smidig 2008 talk)

## Conditionals

Current solution:

```
if (!isInvalid(value)) {
   ...
}
```

Negatives are harder to understand than positives.

Possible improvement:

```
if (isValid(value)) {
   ...
}
```

but also

cellent

I agree with most of the stuff I present here...

(facsimile from my Smidig 2008 talk)

## Explanatory variables

Current solution:

```
boolean isLeapYear(int year) {
  return ((year % 4 == 0) && (year % 400 == 0)) ||
         ((year % 4 == 0) && (year % 100 != 0));
}
```

The code above does the right thing, but it is possible to improve the readability.

Possible improvement:

```
boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundreth = year % 100 == 0;
  boolean fourHundreth = year % 400 == 0;
  return fourth && (!hundreth || fourHundreth);
}
```

I agree with most of the stuff I present here...

# (facsimile from my Smidig 2008 talk)

## Functions

A function should ideally do just one thing, and do it well. Above is an example of a function that does many things. If a function does more than one thing, consider splitting it.

Current solution:

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

Possible improvement:

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

I agree with most of the stuff I present here...

# Which code snippet is better?

# Which code snippet is better?

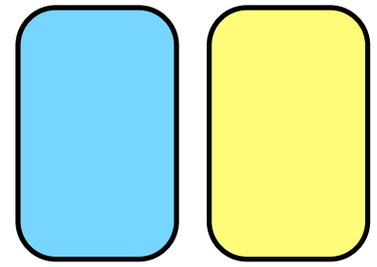
```
if (!isInvalid(value)) {
   ...
}
```

# Which code snippet is better?

```
if (isValid(value)) {
   ...
}
```

```
if (!isInvalid(value)) {
   ...
}
```

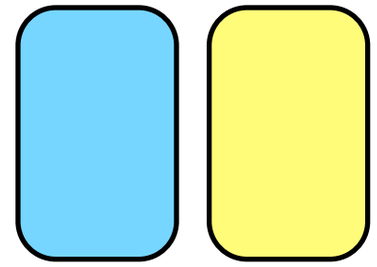
# Which code snippet is better?



```
if (isValid(value)) {
   ...
}
```

```
if (!isInvalid(value)) {
   ...
}
```

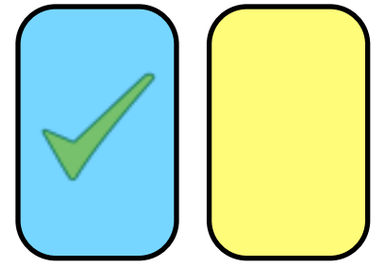
# Which code snippet is better?



```
if (isValid(value)) {
   ...
}
```

```
if (!isInvalid(value)) {
   ...
}
```

# Which code snippet is better?



```
if (isValid(value)) {
   ...
}
```

```
if (!isInvalid(value)) {
   ...
}
```

# Which code snippet is better?

# Which code snippet is better?

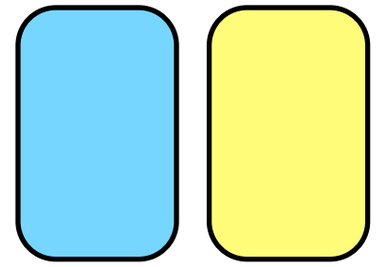
```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

# Which code snippet is better?

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

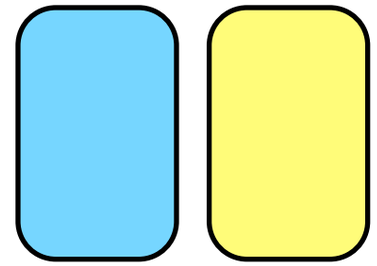
# Which code snippet is better?



```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

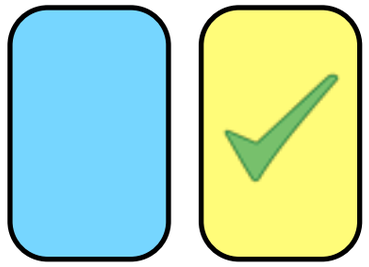
# Which code snippet is better?



```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```


```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

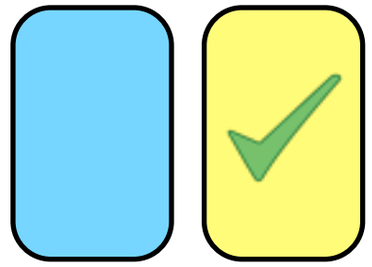
# Which code snippet is better?



```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```


```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

# Which code snippet is better?

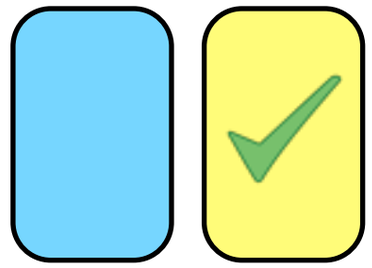


```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```


better?

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

# Which code snippet is better?



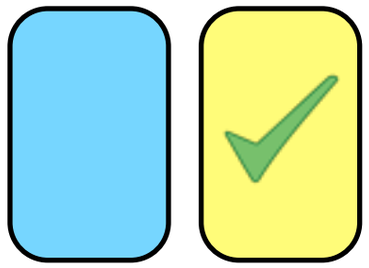
```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```


better?

cleaner?

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

# Which code snippet is better?

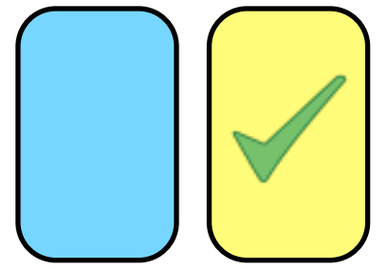


```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```


better?
cleaner?
style?

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

# Which code snippet is better?



```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```


better?
cleaner?
style?
stable code?

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

Which code snippet is ~~better~~ more stable?

# Which code snippet is ~~better~~ more stable?

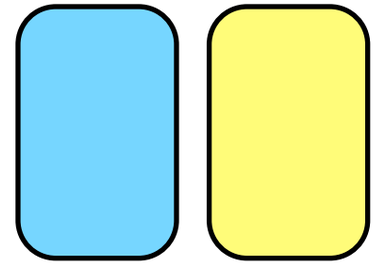
```
boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundreth = year % 100 == 0;
  boolean fourHundreth = year % 400 == 0;
  return fourth && (!hundreth || fourHundreth);
}
```

# Which code snippet is ~~better~~ more stable?

```
boolean isLeapYear(int year) {
  return ((year % 4 == 0) && (year % 400 == 0)) ||
         ((year % 4 == 0) && (year % 100 != 0));
}
```

```
boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundreth = year % 100 == 0;
  boolean fourHundreth = year % 400 == 0;
  return fourth && (!hundreth || fourHundreth);
}
```

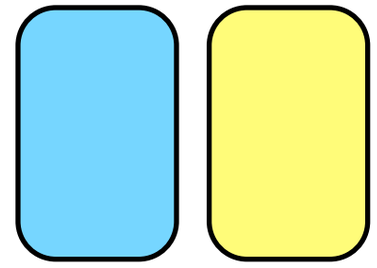
# Which code snippet is ~~better~~ more stable?



```
boolean isLeapYear(int year) {
  return ((year % 4 == 0) && (year % 400 == 0)) ||
         ((year % 4 == 0) && (year % 100 != 0));
}
```

```
boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundreth = year % 100 == 0;
  boolean fourHundreth = year % 400 == 0;
  return fourth && (!hundreth || fourHundreth);
}
```

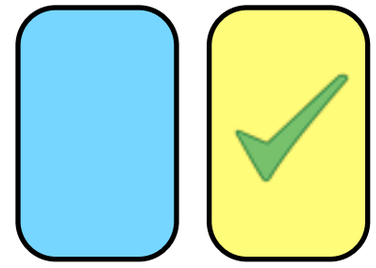
# Which code snippet is ~~better~~ more stable?



```
boolean isLeapYear(int year) {
  return ((year % 4 == 0) && (year % 400 == 0)) ||
         ((year % 4 == 0) && (year % 100 != 0));
}
```

```
boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundreth = year % 100 == 0;
  boolean fourHundreth = year % 400 == 0;
  return fourth && (!hundreth || fourHundreth);
}
```

# Which code snippet is ~~better~~ more stable?



```
boolean isLeapYear(int year) {
  return ((year % 4 == 0) && (year % 400 == 0)) ||
         ((year % 4 == 0) && (year % 100 != 0));
}
```

```
boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundreth = year % 100 == 0;
  boolean fourHundreth = year % 400 == 0;
  return fourth && (!hundreth || fourHundreth);
}
```

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

```
if (!isInvalid(value)) {
  ...
}
```

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

```
if (!isInvalid(value)) {
  ...
}
```

17%

```
if (isValid(value)) {
  ...
}
```

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

```
if (!isInvalid(value)) {
  ...
}
```

```
if (isValid(value)) {
  ...
}
```

17%

```
if (isValid(value)) {
  ...
}
```

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

```
if (!isInvalid(value)) {
  ...
}
```

17%

```
if (isValid(value)) {
  ...
}
```

```
if (isValid(value)) {
  ...
}
```

0.04%

```
if (!isInvalid(value)) {
  ...
}
```

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

```
if (!isInvalid(value)) {
  ...
}
```

17%

```
if (isValid(value)) {
  ...
}
```

```
if (isValid(value)) {
  ...
}
```

0.04%

```
if (!isInvalid(value)) {
  ...
}
```

ie,

```
if (!isInvalid(value)) {
  ...
}
```

is more unstable than

```
if (isValid(value)) {
  ...
}
```

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

```
if (!isInvalid(value)) {
  ...
}
```

17%

```
if (isValid(value)) {
  ...
}
```

```
if (isValid(value)) {
  ...
}
```

0.04%

```
if (!isInvalid(value)) {
  ...
}
```

ie,

```
if (!isInvalid(value)) {
  ...
}
```

is more unstable than

```
if (isValid(value)) {
  ...
}
```

Is this a useful way of thinking about code?

Is it possible to say, when comparing two equivalent pieces of code, that one snippet is more likely to change into the other snippet, than vice versa?

For example:

```
if (!isInvalid(value)) {
  ...
}
```

17% →

```
if (isValid(value)) {
  ...
}
```

```
if (isValid(value)) {
  ...
}
```

0.04% →

```
if (!isInvalid(value)) {
  ...
}
```

ie,

```
if (!isInvalid(value)) {
  ...
}
```

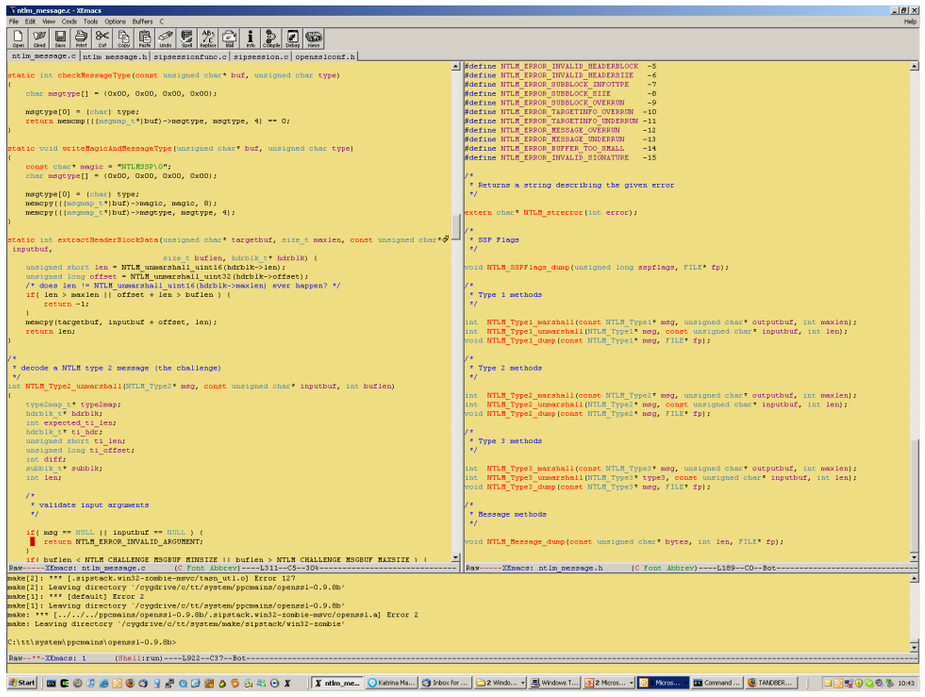
is more unstable than

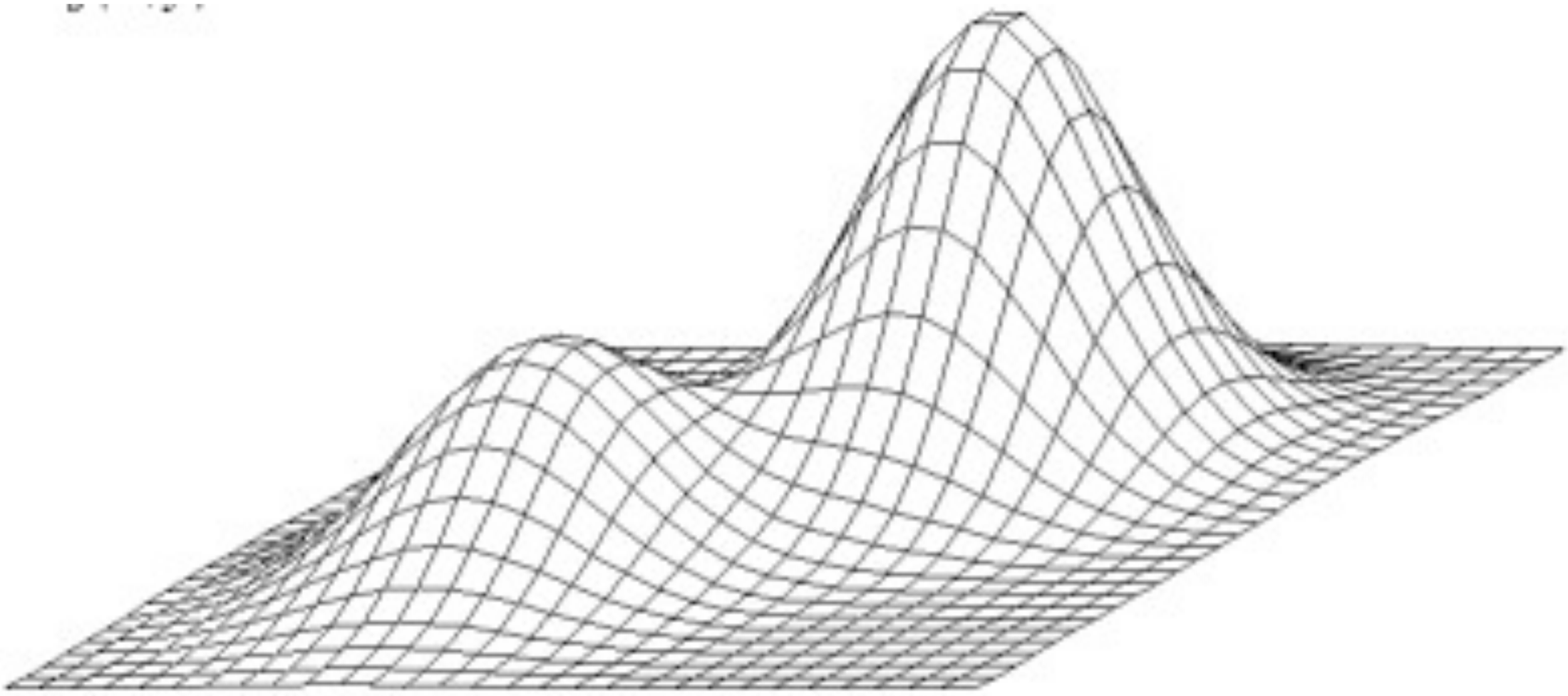
```
if (isValid(value)) {
  ...
}
```

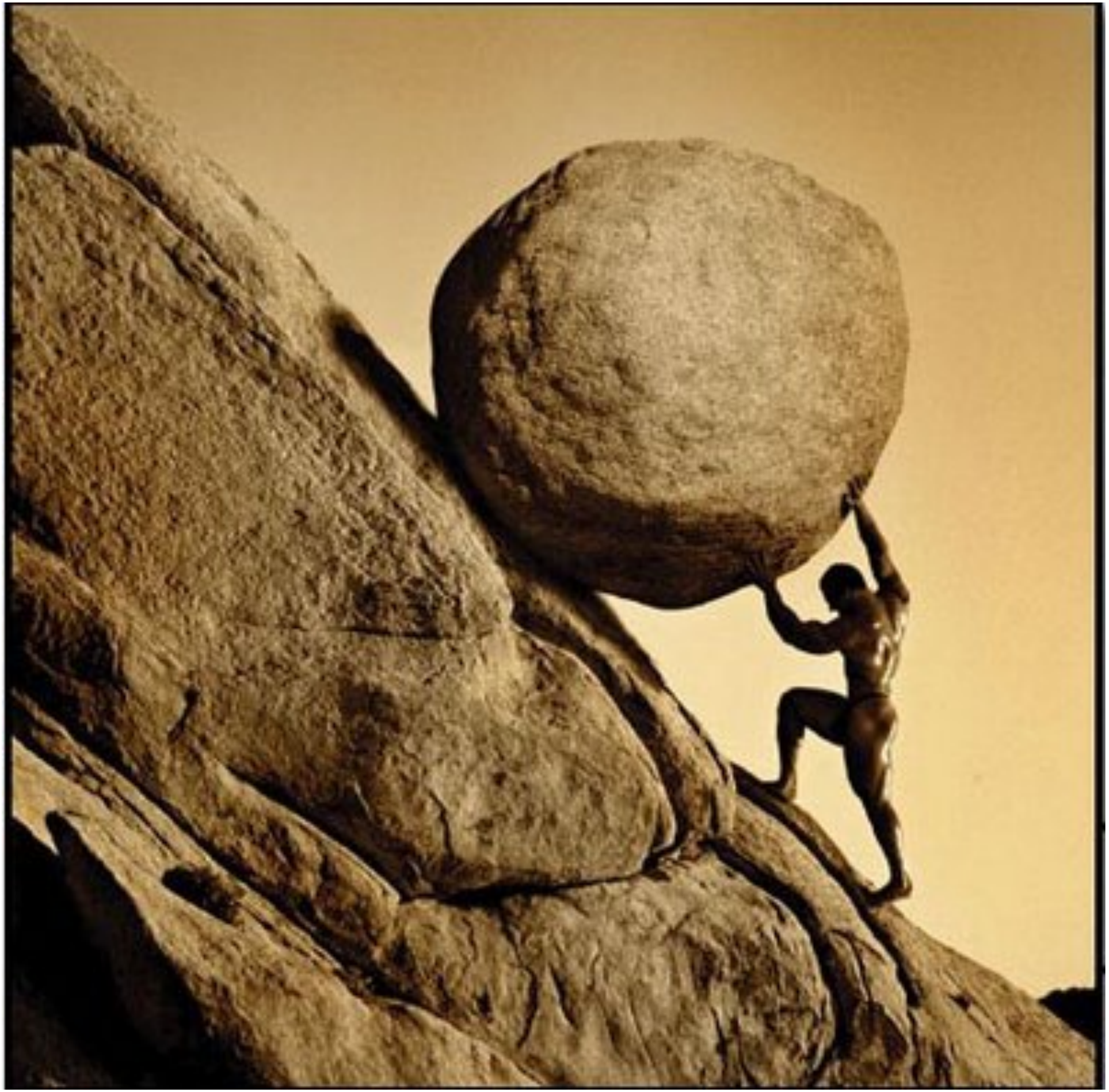
Is this a useful way of thinking about code?

If so, does it make sense to talk about code entropy?

# Entropy? Is there a link between software development and thermodynamics?





# (Facsimile from Code Archaeology talk at ACCU 2010)

## Code Archaeology - Real stories from a real codebase
by Jon Jagger and Olve Maudal

Nothing beats a large codebase that has been worked on and cared for by hundreds of developers over many years, and that is still in good shape, and that can still be used to churn out one successful product after the other.

We have studied such a codebase; we have analysed and commented on actual changes done by professional programmers over the years. In particular we have paid interest to the small refactoring tasks and code cleaning activities that seems to be needed to keep the codebase in good shape - the small and "insignificant" changes that professionals do to avoid rot in the codebase.

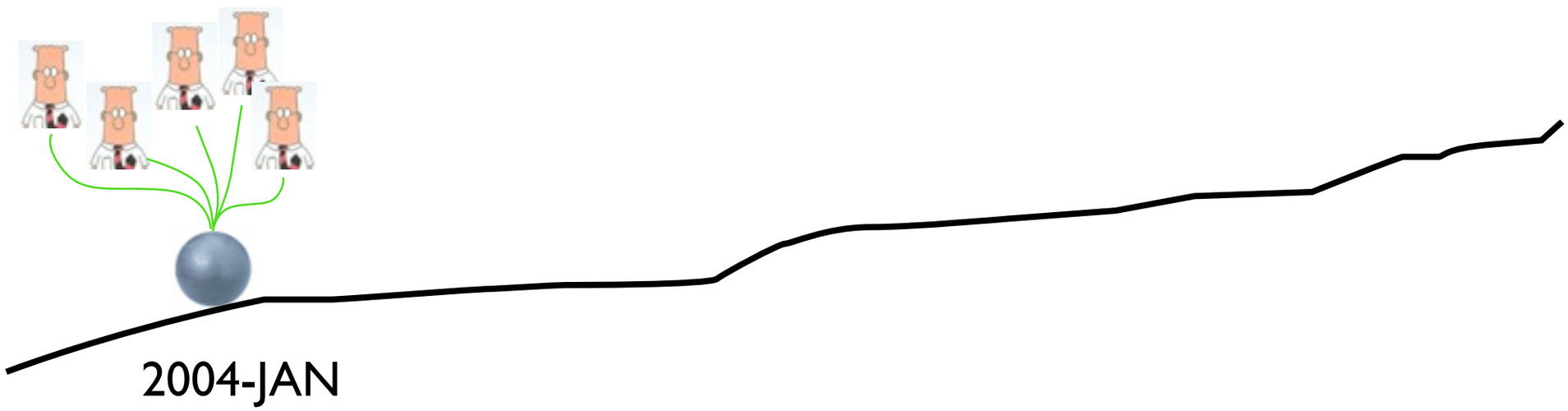
There will be a lot of C ~~and C++~~ code in this talk. We will focus on the small details of code cleaning. Be prepared for tough discussions about what really adds value or not to a codebase.

The story of

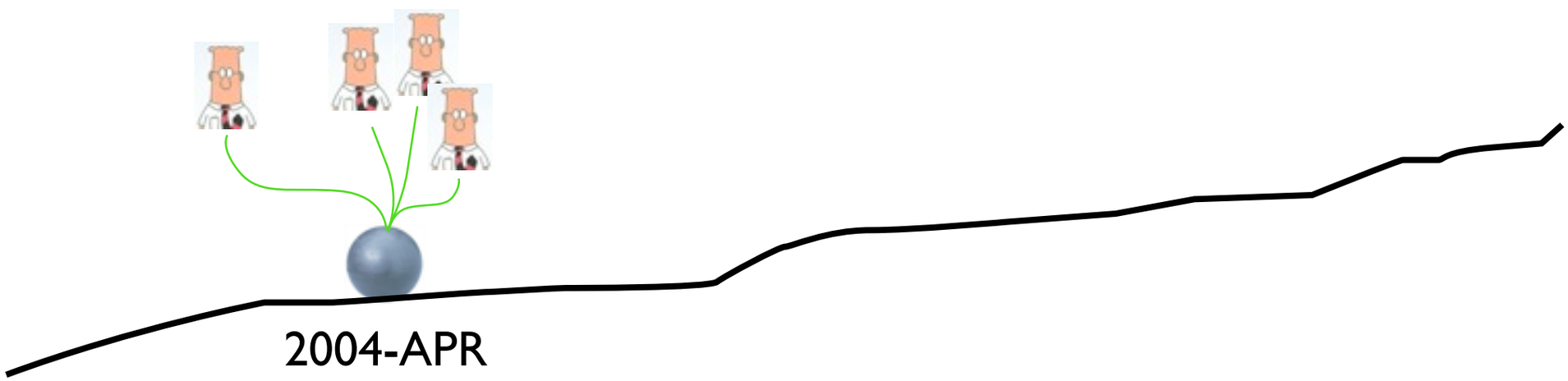
`RAS_getCallByCallId()`

(2004-)

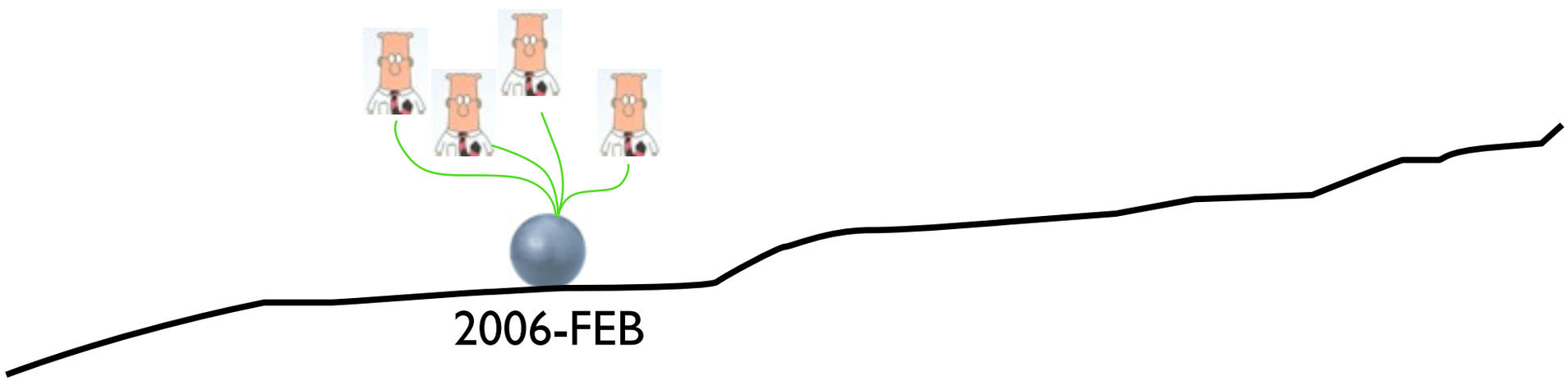
```
CALL *RAS_getCallByCallId(PROC_DATA *pProc, CALLID iCallId) {
    int i;

    for (i=0; i<H323CC_HIGHEST_IND; i++) {
        if(fsm_pIData->calls[i].bInUse && fsm_pIData->calls[i].iCallId == iCallId)
            return &fsm_pIData->calls[i];
    }
    return NULL;
}
```



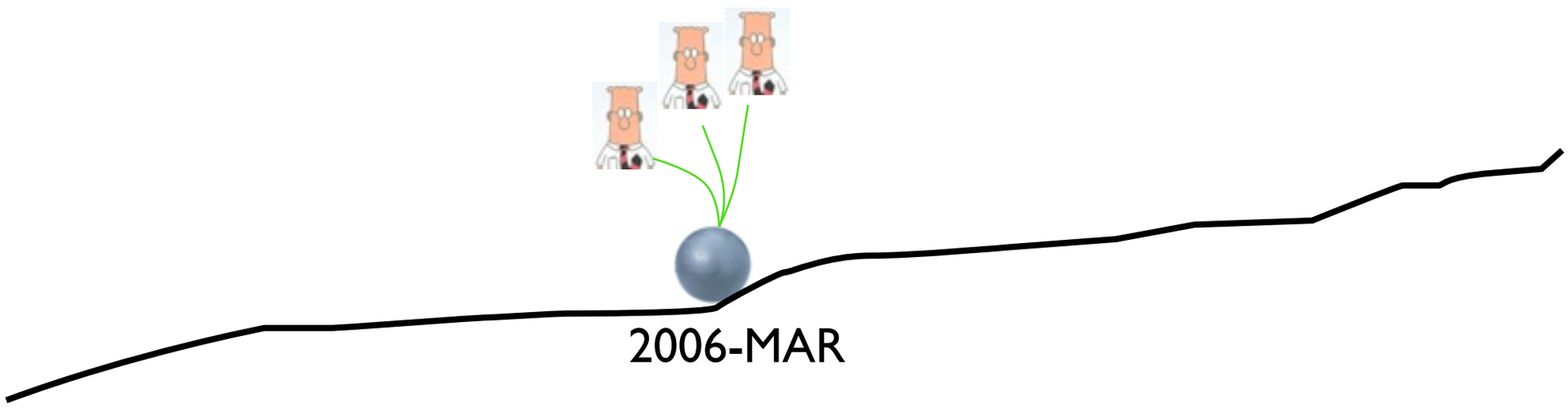
```
CALL *RAS_getCallByCallId(PROC_DATA *pProc, CALLID iCallId) {
    int i;

    for (i=0; i<RAS_MAX_CALLS; i++) {
        if(fsm_pIData->calls[i].bInUse && fsm_pIData->calls[i].iCallId == iCallId)
            return &fsm_pIData->calls[i];
    }
    return NULL;
}
```



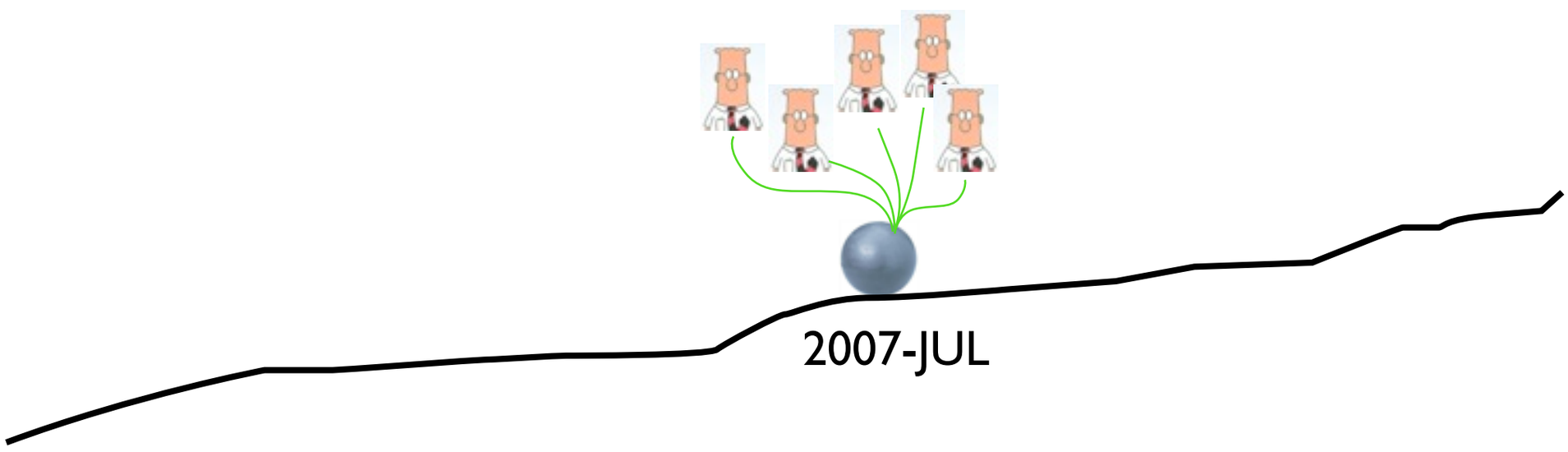
```
CALL *RAS_getCallByCallId(PROC_DATA *pProc, CALLID iCallId)
{
    int i;

    for (i=0; i<RAS_MAX_CALLS; i++) {
        if(fsm_pIData->calls[i].bInUse && fsm_pIData->calls[i].iCallId == iCallId)
            return &fsm_pIData->calls[i];
    }
    return NULL;
}
```



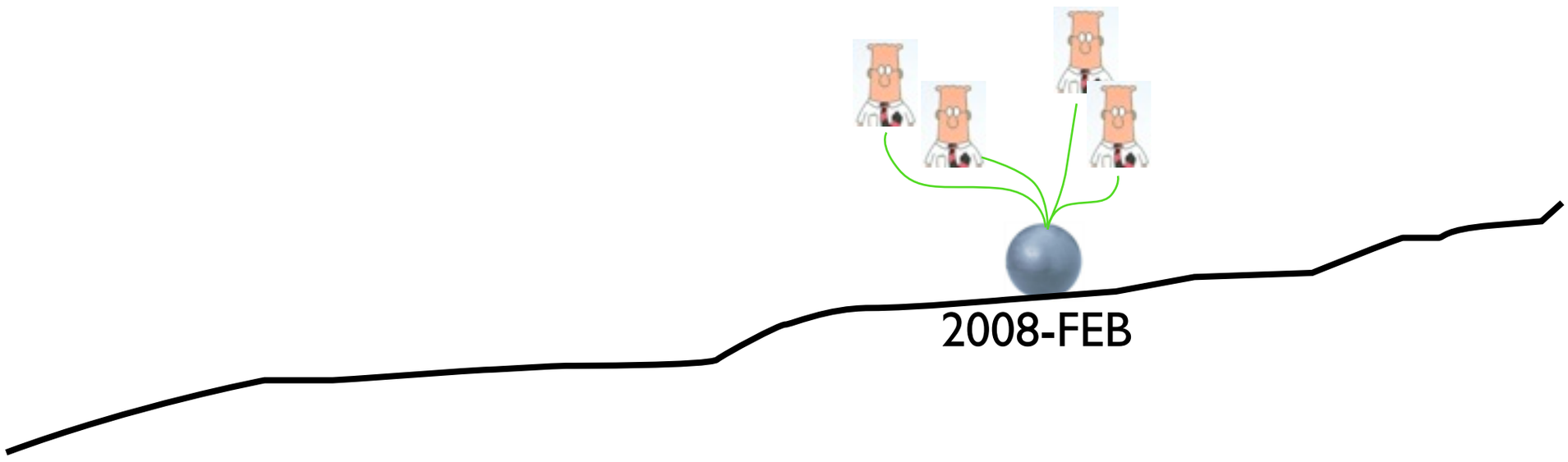
```
CALL *RAS_getCallByCallId(PROC_DATA *pProc, CALLID iCallId)
{
    int i;

    for (i=0; i < fsm_pIData->rasInit.iMaxCalls; i++) {
        if(fsm_pIData->calls[i].bInUse && fsm_pIData->calls[i].iCallId == iCallId)
            return &fsm_pIData->calls[i];
    }
    return NULL;
}
```



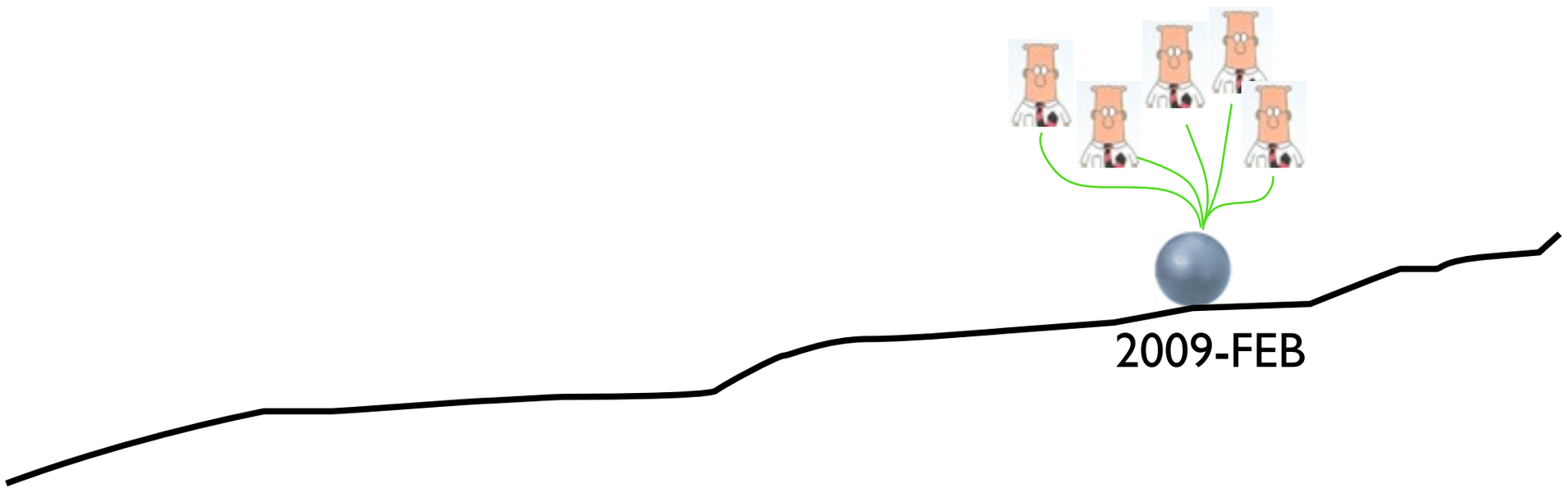
```
CALL *RAS_getCallByCallId(PROC_DATA *pProc, CALLID callid)
{
    int i;

    for (i = 0; i < fsm_pIData->rasInit.iMaxCalls; i++) {
        if (fsm_pIData->calls[i].inUse && fsm_pIData->calls[i].callid == callid) {
            return &fsm_pIData->calls[i];
        }
    }
    return NULL;
}
```



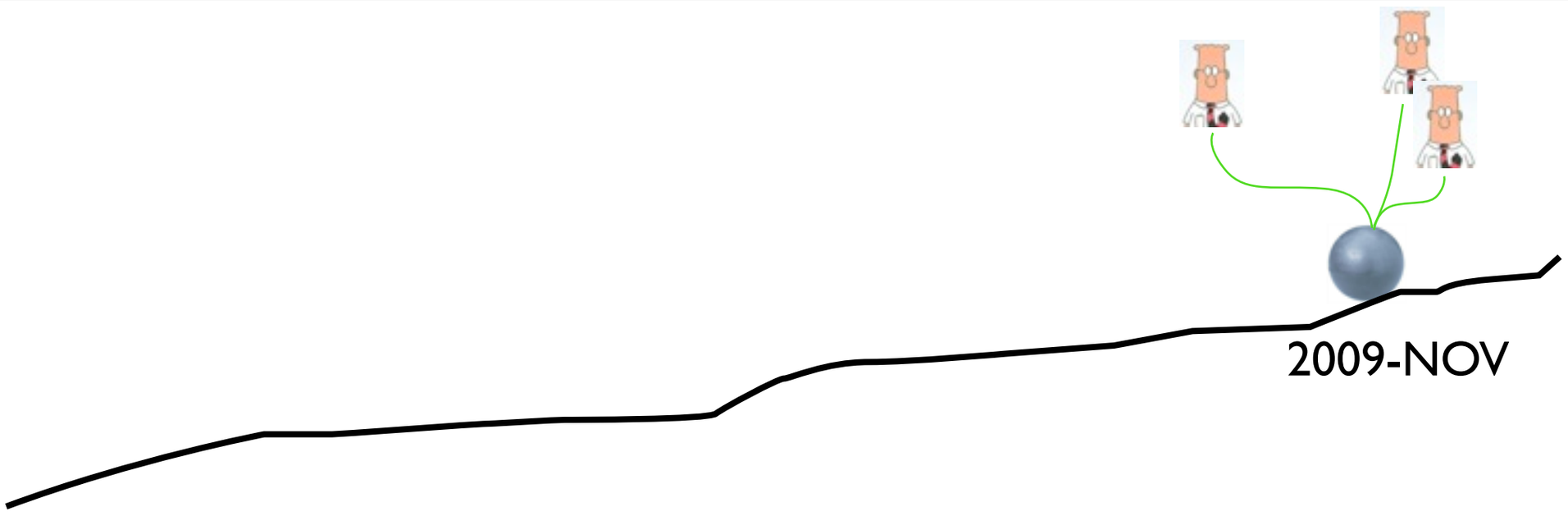
```
CALL *RAS_getCallByCallId(PROC_DATA *pProc, CALLID callid)
{
    int i;

    for (i = 0; i < fsm_pIData->rasInit.maxCalls; i++) {
        if (fsm_pIData->calls[i].inUse && fsm_pIData->calls[i].callid == callid) {
            return &fsm_pIData->calls[i];
        }
    }
    return NULL;
}
```



```
CALL * RAS_getCallByCallId(struct RAS_PRIV * priv,
                           CALLID callid)
{
    int i;
    for (i = 0; i < priv->rasInit.maxCalls; i++) {
        if (priv->calls[i].inUse && priv->calls[i].callid == callid) {
            return &priv->calls[i];
        }
    }
    return NULL;
}
```



```
struct CALL * RAS_getCallByCallId(struct RAS_PRIV * priv,
                                  CALLID callid)
{
    int i;
    for (i = 0; i < priv->rasInit.maxCalls; i++) {
        if (priv->calls[i].inUse && priv->calls[i].callid == callid) {
            return &priv->calls[i];
        }
    }
    return NULL;
}
```



# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

- Readability

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

- Readability
- Dependency management

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

- Readability
- Dependency management
- Using new language features

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

- Readability
- Dependency management
- Using new language features
- Collective ownership

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

- Readability
- Dependency management
- Using new language features
- Collective ownership
- Idiom based programming

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on “robust” layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

- Readability
- Dependency management
- Using new language features
- Collective ownership
- Idiom based programming
- Reducing scope of variables

# Some observations from this particular codebase:

- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on "robust" layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

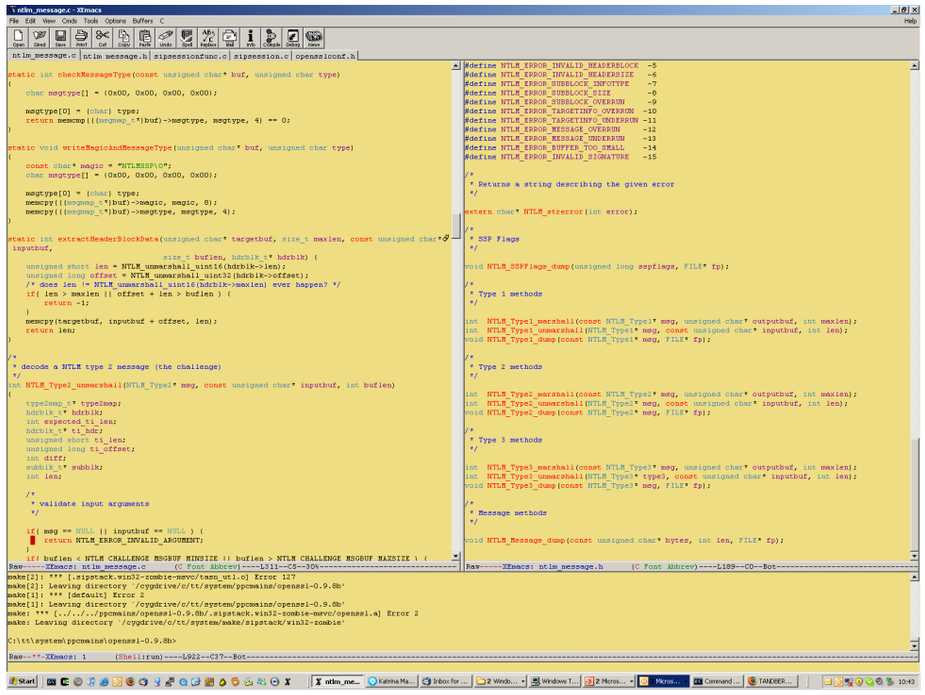
- Readability
- Dependency management
- Using new language features
- Collective ownership
- Idiom based programming
- Reducing scope of variables
- Tabs always looses for spaces

# Some observations from this particular codebase:

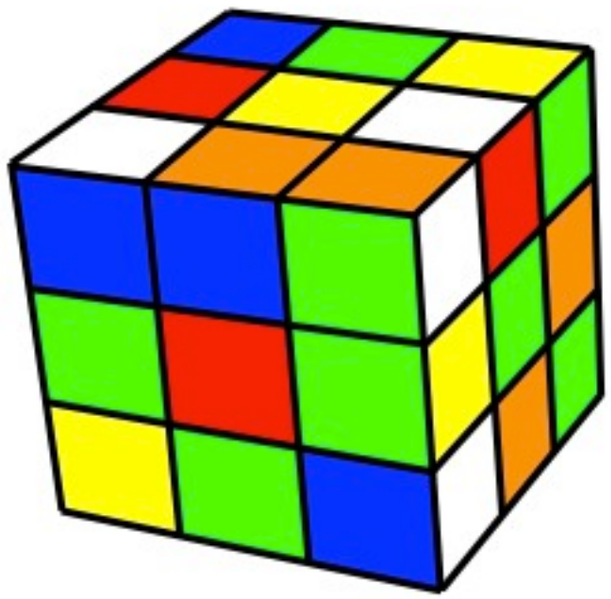
- more focus on readability
- work to reduce compilation time
- dependency analysis / control / management
- less use of inline functions
- dependency injection
- more use of const, size_t and assert
- more use of forward declarations
- typedef struct deprecated
- moving away from corporate libraries
- using new language featuers, eg C99
- focus on standard C
- prefer C over C++, only use C++ where necessary
- proper casting, eg, `len < (int)sizeof x` vs `(size_t)len < sizeof x`
- macros gradually replaced, less use of preprocessor
- peer reviews / pair programming / patches mailing list
- collective ownership
- towards idioms
- change functions to return void when callers not care about the return value
- xx_assert() replaced with assert()
- reducing scope of variables
- abbreviations replaced with more descriptive names
- explanation variables
- initialization of variables
- for (int i=0; i<42; i++) vs for (int i=0; i!=42; ++i)
- reduce use of fix ints (uint32_t -> int)
- order of include files
- early returns, less nesting
- intention revealing typenames (eg, WORD -> bool)
- less use of NULL
- log messages tend to be removed
- dehungarization
- decamelization
- aligned braces for functions, disaligned for if/for/while
- {} removed from single line blocks
- removing parenthesis
- long lines are broken into 80 character lines
- tabs are replaced with spaces
- char * s seems to be more stable than K&R and BS style
- In C, post-increment seems to be more stable than pre-increment (i++ vs ++i)
- focus on "robust" layout
- increased horizontal and vertical spacing
- indentation 4 spaces
- space around operators
- block comments are removed
- removing comments by improving code

- Readability
- Dependency management
- Using new language features
- Collective ownership
- Idiom based programming
- Reducing scope of variables
- Tabs always looses for spaces
- Removing comments and improve code

# Entropy

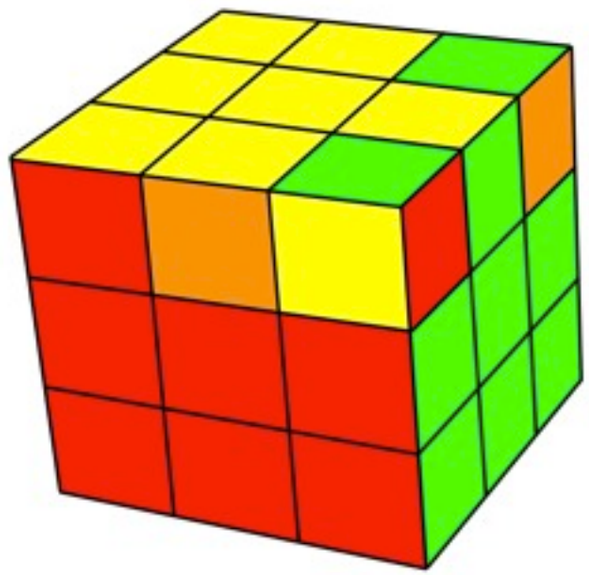

ntlm_message.c - XEmacs
File Edit View Cmds Tools Options Buffers C Help
ntlm_message.c | ntlm message.h | sipsessionfunc.c | sipsession.c | opensslconf.h
static int checkMessageType(const unsigned char* buf, unsigned char type)
static void writeMagicAndMessageType(unsigned char* buf, unsigned char type)
static int extractHeaderBlockData(unsigned char* targetbuf, size_t maxlen, const unsigned char*
inputbuf, size_t buflen, hdrblk_t* hdrblk) {
/*
* decode a NTLM type 2 message (the challenge)
*/
int NTLM_Type2_unmarshall(NTLM_Type2* msg, const unsigned char* inputbuf, int buflen)
/* validate input arguments
*/
if( msg == NULL || inputbuf == NULL ) {
return NTLM_ERROR_INVALID_ARGUMENT;
}
if( buflen < NTLM_CHALLENGE_MSGBUF_MINSIZE || buflen > NTLM_CHALLENGE_MSGBUF_MAXSIZE ) {
#define NTLM_ERROR_INVALID_HEADERBLOCK -5
#define NTLM_ERROR_INVALID_HEADERSIZE -6
#define NTLM_ERROR_SUBBLOCK_INFOTYPE -7
#define NTLM_ERROR_SUBBLOCK_SIZE -8
#define NTLM_ERROR_SUBBLOCK_OVERRUN -9
#define NTLM_ERROR_TARGETINFO_OVERRUN -10
#define NTLM_ERROR_TARGETINFO_UNDERRUN -11
#define NTLM_ERROR_MESSAGE_OVERRUN -12
#define NTLM_ERROR_MESSAGE_UNDERRUN -13
#define NTLM_ERROR_BUFFER_TOO_SMALL -14
#define NTLM_ERROR_INVALID_SIGNATURE -15
/*
* Returns a string describing the given error
*/
extern char* NTLM_strerror(int error);
/*
* SSP Flags
*/
void NTLM_SSPFlags_dump(unsigned long sspflags, FILE* fp);
/*
* Type 1 methods
*/
int NTLM_Type1_marshall(const NTLM_Type1* msg, unsigned char* outputbuf, int maxlen);
int NTLM_Type1_unmarshall(NTLM_Type1* msg, const unsigned char* inputbuf, int len);
void NTLM_Type1_dump(const NTLM_Type1* msg, FILE* fp);
/*
* Type 2 methods
*/
int NTLM_Type2_marshall(const NTLM_Type2* msg, unsigned char* outputbuf, int maxlen);
int NTLM_Type2_unmarshall(NTLM_Type2* msg, const unsigned char* inputbuf, int len);
void NTLM_Type2_dump(const NTLM_Type2* msg, FILE* fp);
/*
* Type 3 methods
*/
int NTLM_Type3_marshall(const NTLM_Type3* msg, unsigned char* outputbuf, int maxlen);
int NTLM_Type3_unmarshall(NTLM_Type3* type3, const unsigned char* inputbuf, int len);
void NTLM_Type3_dump(const NTLM_Type3* msg, FILE* fp);
/*
* Message methods
*/
void NTLM_Message_dump(const unsigned char* bytes, int len, FILE* fp);
Raw-----XEmacs: ntlm_message.c (C Font Abbrev)----L311--C5--30%
Raw-----XEmacs: ntlm_message.h (C Font Abbrev)----L189--C0--Bot
make[2]: *** [.sipstack.win32-zombie-msvc/tasn_utl.o] Error 127
make[2]: Leaving directory '/cygdrive/c/tt/system/ppcmains/openssl-0.9.8b'
make[1]: *** [default] Error 2
make[1]: Leaving directory '/cygdrive/c/tt/system/ppcmains/openssl-0.9.8b'
make: *** [../../ppcmains/openssl-0.9.8b/.sipstack.win32-zombie-msvc/openssl.a] Error 2
make: Leaving directory '/cygdrive/c/tt/system/make/sipstack/win32-zombie'
C:\tt\system\ppcmains\openssl-0.9.8b>
Raw--**-XEmacs: 1 (Shell:run)----L922--C37--Bot
Start
ntlm_me...
Katrina Ma...
Inbox for ...
2 Windo...
Windows T...
2 Micros...
Micros...
Command ...
TANDBER...
10:43

# Entropy



high entropy

>



low entropy

Entropy is a measure of how organized or disorganized a system is

(wikipedia)

# Code Entropy

# Code Entropy

*Consider two semantically similar code snippets, A and B.*

# Code Entropy

*Consider two semantically similar code snippets, A and B.*

```
  ...
  if (!is_open(socket))
    return false;
  else
    return true;
}
```

A

# Code Entropy

*Consider two semantically similar code snippets, A and B.*

```
  ...
  if (!is_open(socket))
    return false;
  else
    return true;
}
```

A

```
  ...
  if (is_open(socket))
    return true;
  else
    return false;
}
```

B

# Code Entropy

*Consider two semantically similar code snippets, A and B.*
*If a group of experts are more likely to change A into B, than vice versa, then code snippet A is less stable.*

```
  ...
  if (!is_open(socket))
    return false;
  else
    return true;
}
```

A

```
  ...
  if (is_open(socket))
    return true;
  else
    return false;
}
```

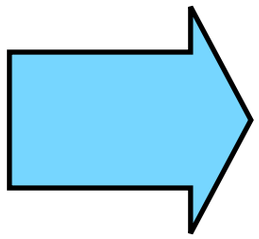
B

# Code Entropy

*Consider two semantically similar code snippets, A and B.*
*If a group of experts are more likely to change A into B, than vice versa, then code snippet A is less stable.*

```
  ...
  if (!is_open(socket))
    return false;
  else
    return true;
}
```

A



```
  ...
  if (is_open(socket))
    return true;
  else
    return false;
}
```

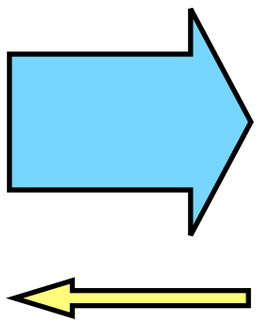
B

# Code Entropy

*Consider two semantically similar code snippets, A and B.*
*If a group of experts are more likely to change A into B, than vice versa, then code snippet A is less stable.*

```
  ...
  if (!is_open(socket))
    return false;
  else
    return true;
}
```

A



```
  ...
  if (is_open(socket))
    return true;
  else
    return false;
}
```

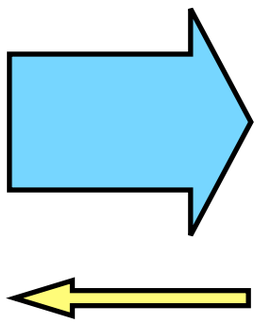
B

# Code Entropy

*Consider two semantically similar code snippets, A and B.*
*If a group of experts are more likely to change A into B, than vice versa, then code snippet A is less stable.*
*Hence, A has higher entropy than B.*

```
  ...
  if (!is_open(socket))
    return false;
  else
    return true;
}
```

A



```
  ...
  if (is_open(socket))
    return true;
  else
    return false;
}
```

B

# Code Entropy

*Consider two semantically similar code snippets, A and B.*
*If a group of experts are more likely to change A into B, than vice versa, then code snippet A is less stable.*
*Hence, A has higher entropy than B.*

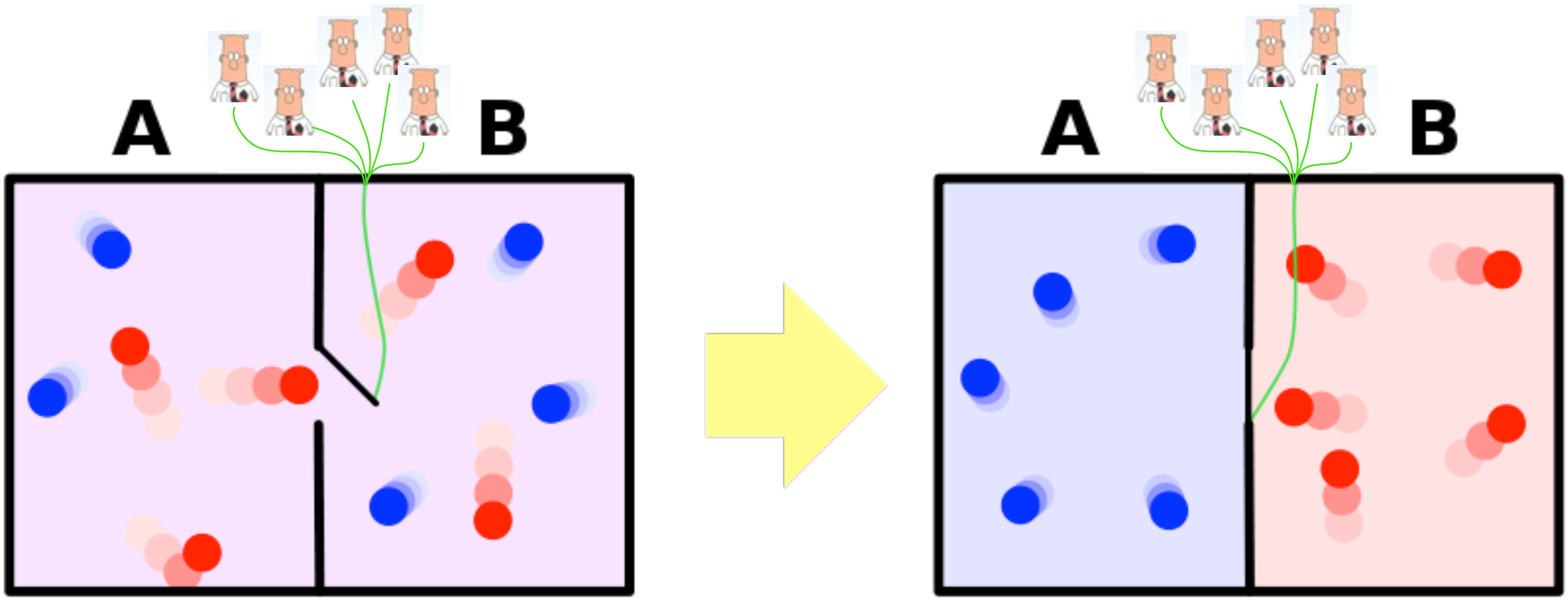
```
  ...
  if (!is_open(socket))
    return false;
  else
    return true;
}
```

```
  ...
  if (is_open(socket))
    return true;
  else
    return false;
}
```

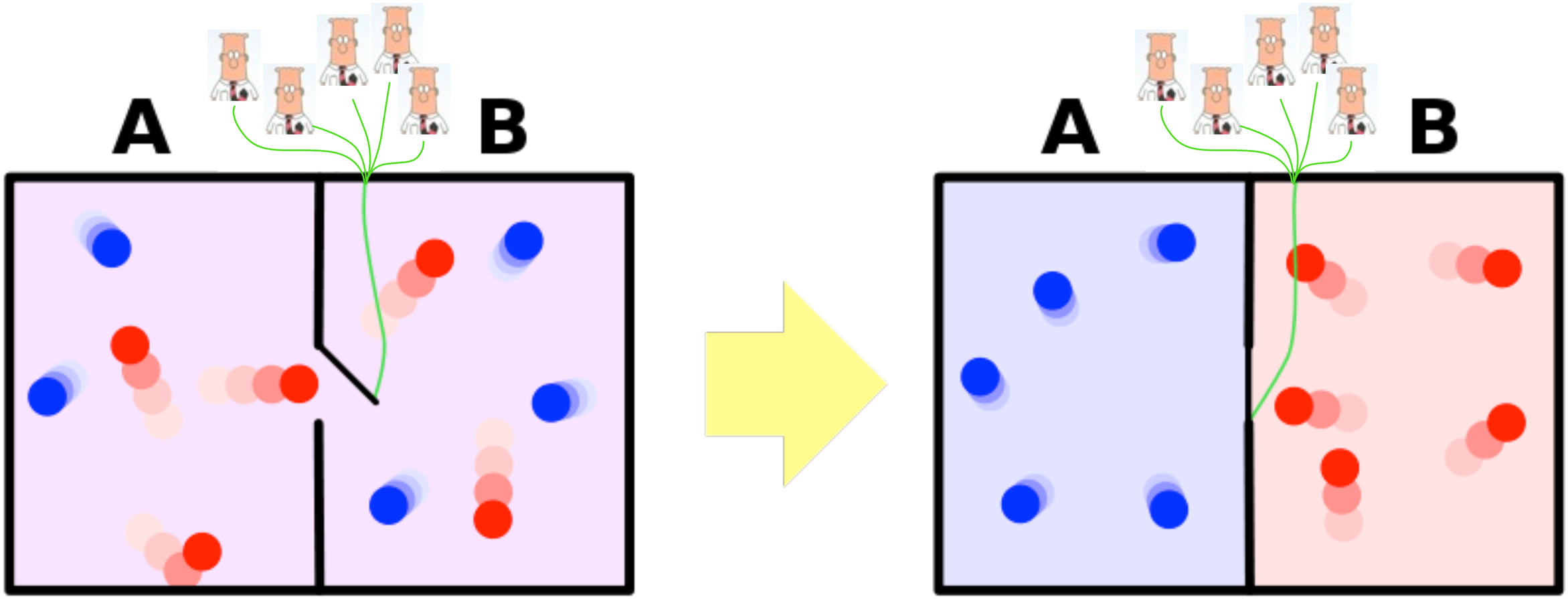
A

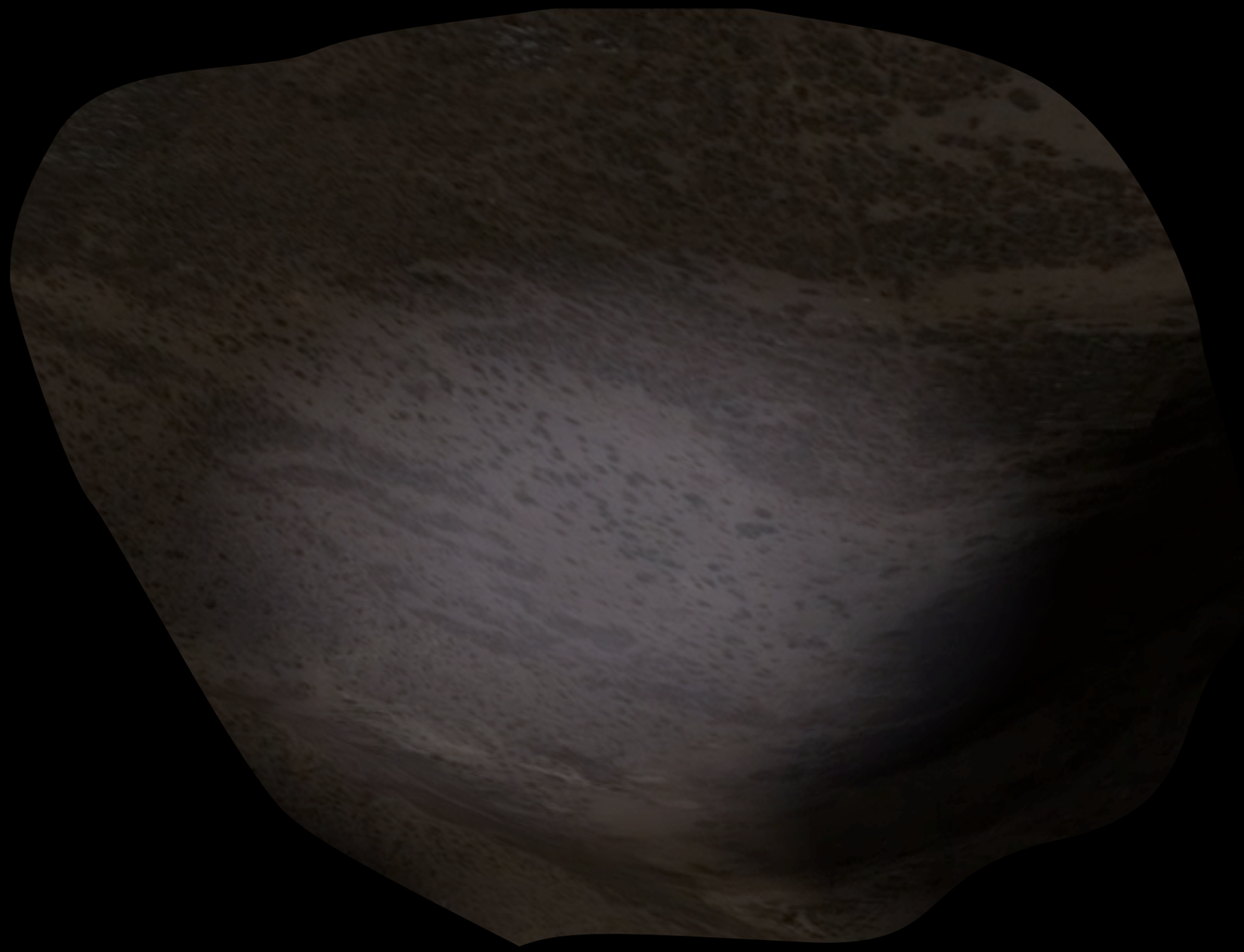
>

B

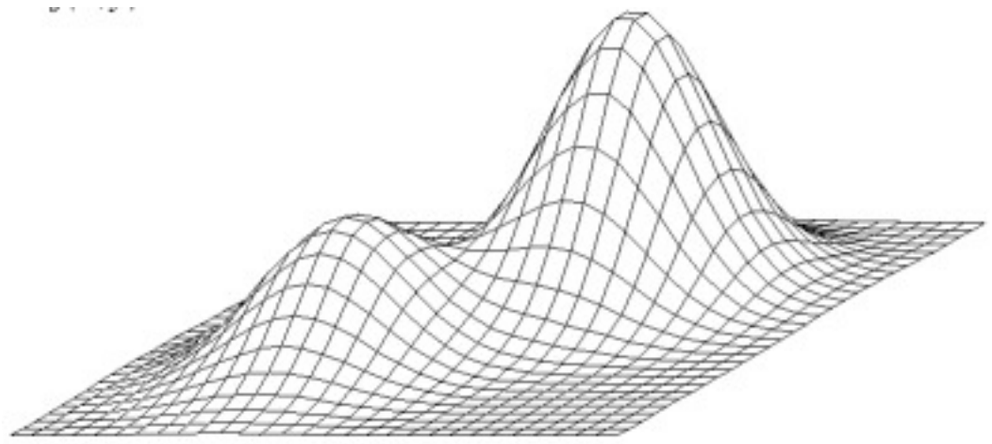

A
B
A
B

# Dilbert's demon

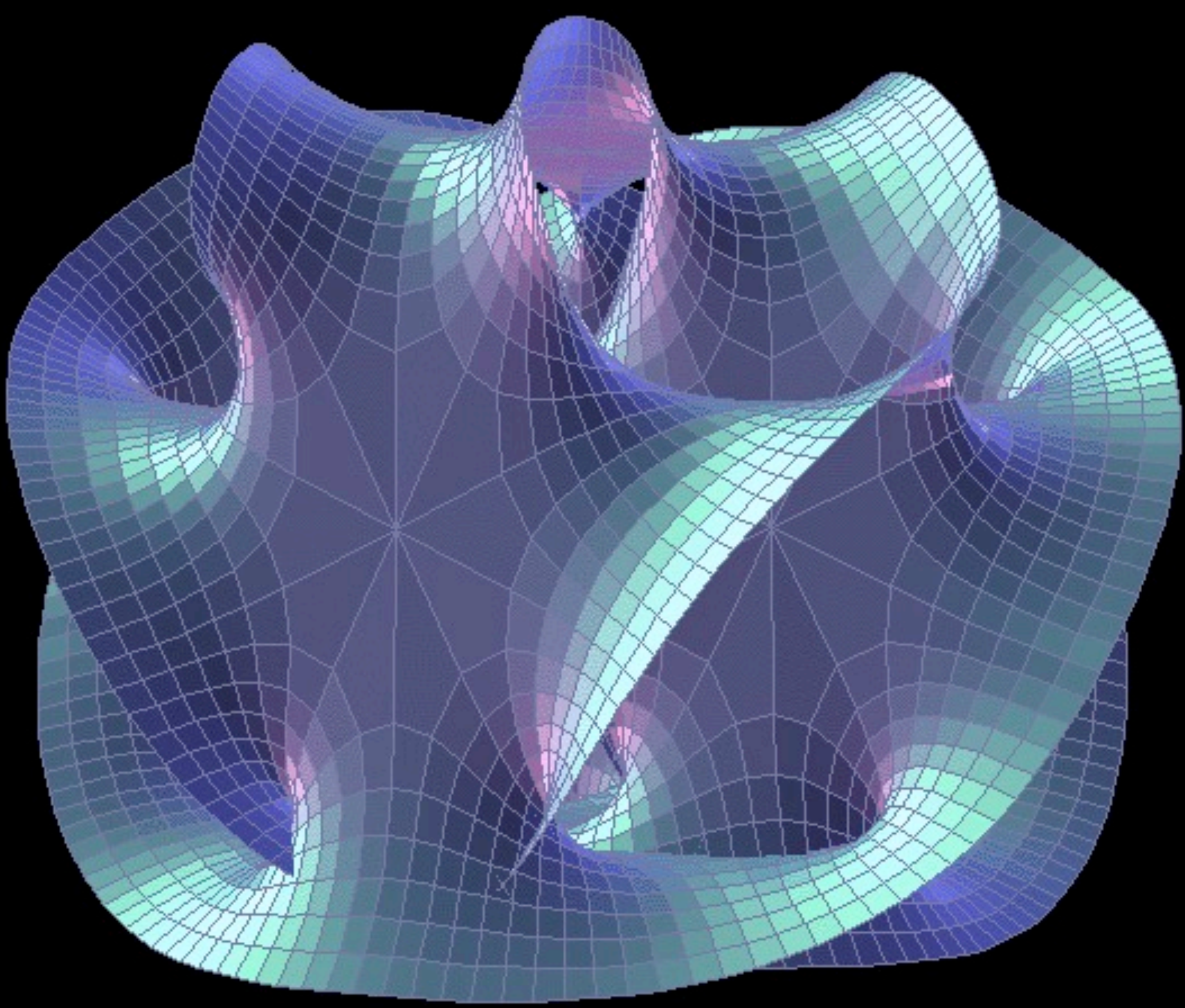


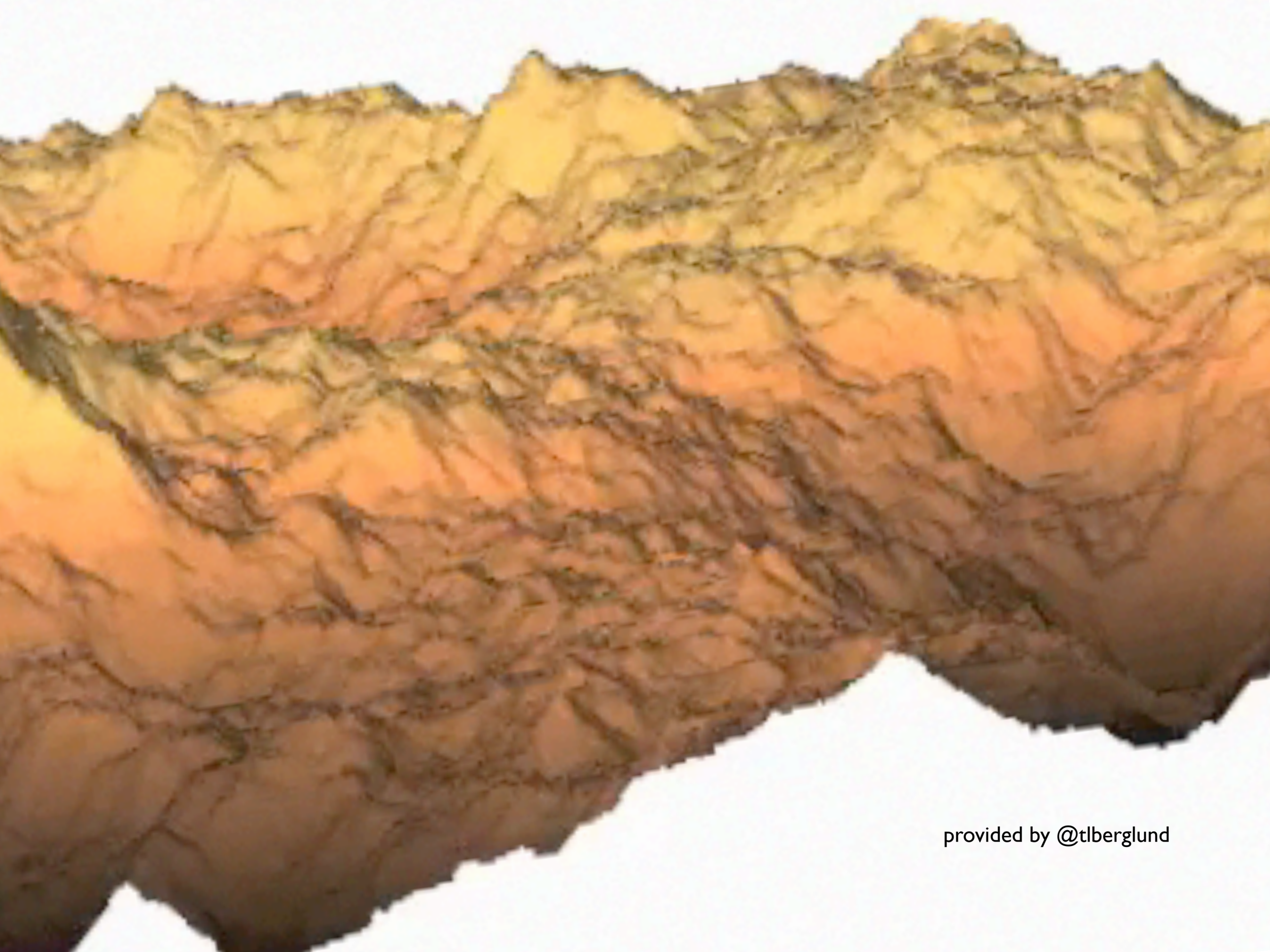


# Optimization problem



hill climbing algorithms
simulated annealing
particle swarm
... and many more

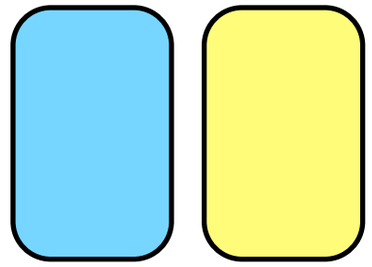




provided by @tlberglund

# Examples

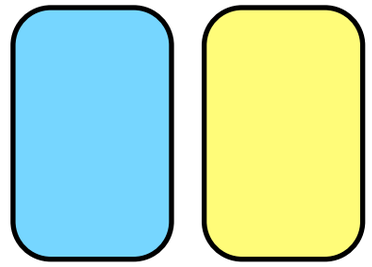
# while vs for loop



```
Enumeration e = pathComponents.elements();
while (e.hasMoreElements() && stream == null) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```

```
InputStream stream = null;

for (Enumeration e = pathComponents.elements(); e.hasMoreElements() && stream == null; ) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```

# while vs for loop

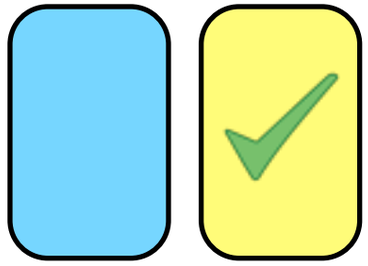


```
Enumeration e = pathComponents.elements();
while (e.hasMoreElements() && stream == null) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```

```
InputStream stream = null;

for (Enumeration e = pathComponents.elements(); e.hasMoreElements() && stream == null; ) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```


AntClassLoader.java

# while vs for loop

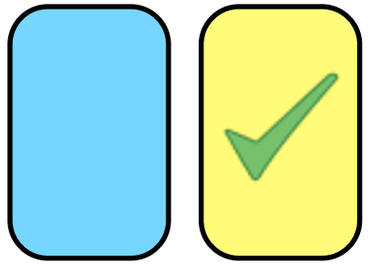


```
Enumeration e = pathComponents.elements();
while (e.hasMoreElements() && stream == null) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```

```
InputStream stream = null;

for (Enumeration e = pathComponents.elements(); e.hasMoreElements() && stream == null; ) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```

AntClassLoader.java

# while vs for loop

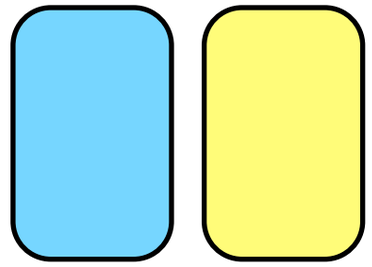


```
Enumeration e = pathComponents.elements();
while (e.hasMoreElements() && stream == null) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```

```
InputStream stream = null;

for (Enumeration e = pathComponents.elements(); e.hasMoreElements() && stream == null; ) {
    File pathComponent = (File)e.nextElement();
    stream = getResourceStream(pathComponent, name);
}
return stream;
```

# fashion or trend?


AntClassLoader.java



# order of specifiers

```
private static final int BUFFER_SIZE = 8192;
```

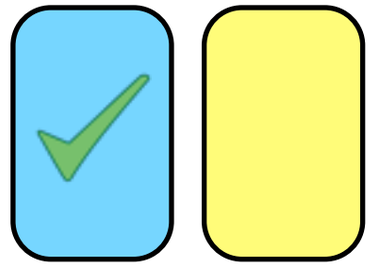
```
static private final int BUFFER_SIZE = 8192;
```

# order of specifiers

```
private static final int BUFFER_SIZE = 8192;
```

```
static private final int BUFFER_SIZE = 8192;
```

AntClassLoader.java

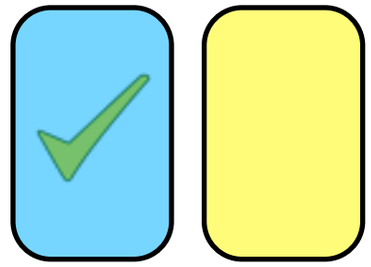


# order of specifiers

```
private static final int BUFFER_SIZE = 8192;
```

```
static private final int BUFFER_SIZE = 8192;
```


AntClassLoader.java



# order of specifiers

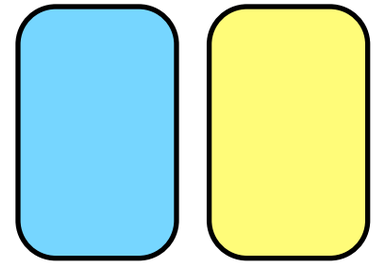
```
private static final int BUFFER_SIZE = 8192;
```

```
static private final int BUFFER_SIZE = 8192;
```

## conform to some standard?


AntClassLoader.java

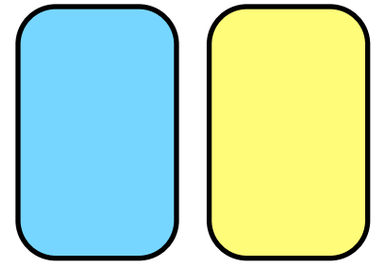
# explicit imports



```
import java.io.*;
import java.util.*;
...
```

```
import java.io.File;
import java.io.IOException;
import java.io.EOFException;
import java.io.InputStream;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Properties;
import java.util.Stack;
import java.util.Vector;
import java.util.Set;
import java.util.HashSet;
import java.util.HashMap;
import java.util.Map;
import java.util.WeakHashMap;
...
```

# explicit imports

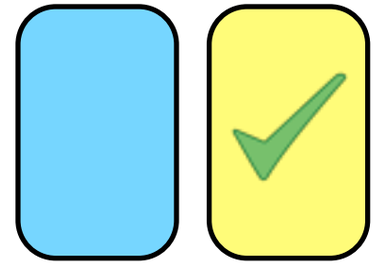


```
import java.io.*;
import java.util.*;
...
```

```
import java.io.File;
import java.io.IOException;
import java.io.EOFException;
import java.io.InputStream;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Properties;
import java.util.Stack;
import java.util.Vector;
import java.util.Set;
import java.util.HashSet;
import java.util.HashMap;
import java.util.Map;
import java.util.WeakHashMap;
...
```


Project.java

# explicit imports

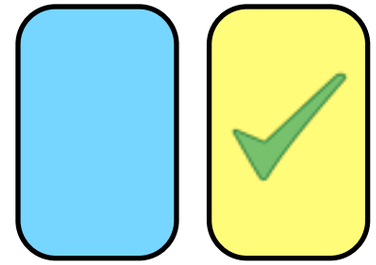


```
import java.io.*;
import java.util.*;
...
```

```
import java.io.File;
import java.io.IOException;
import java.io.EOFException;
import java.io.InputStream;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Properties;
import java.util.Stack;
import java.util.Vector;
import java.util.Set;
import java.util.HashSet;
import java.util.HashMap;
import java.util.Map;
import java.util.WeakHashMap;
...
```


Project.java

# explicit imports



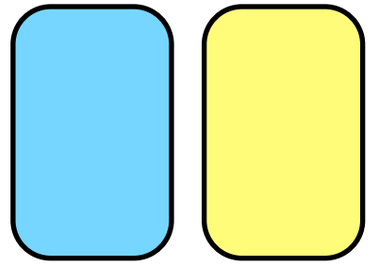
```
import java.io.*;
import java.util.*;
...
```

```
import java.io.File;
import java.io.IOException;
import java.io.EOFException;
import java.io.InputStream;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Properties;
import java.util.Stack;
import java.util.Vector;
import java.util.Set;
import java.util.HashSet;
import java.util.HashMap;
import java.util.Map;
import java.util.WeakHashMap;
...
```

# why? tool support?


Project.java

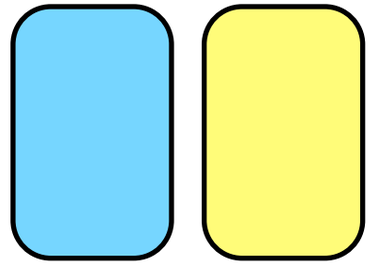
# braces



```
try {
    ...
}
catch(RuntimeException exc) {
    error = exc;
    throw exc;
}
catch(Error err) {
    error = err;
    throw err;
}
finally {
    ...
}
```

```
try {
    ...
} catch(RuntimeException exc) {
    error = exc;
    throw exc;
} catch(Error e) {
    error = e;
    throw e;
} finally {
    ...
}
```

# braces

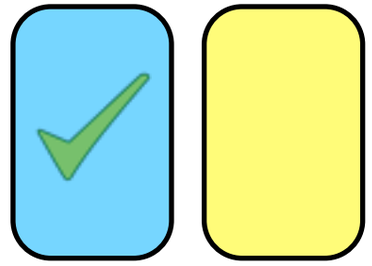


```
try {
    ...
}
catch(RuntimeException exc) {
    error = exc;
    throw exc;
}
catch(Error err) {
    error = err;
    throw err;
}
finally {
    ...
}
```

```
try {
    ...
} catch(RuntimeException exc) {
    error = exc;
    throw exc;
} catch(Error e) {
    error = e;
    throw e;
} finally {
    ...
}
```


Project.java

# braces

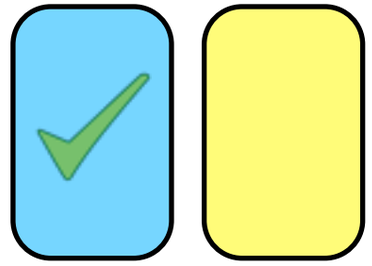


```
try {
    ...
}
catch(RuntimeException exc) {
    error = exc;
    throw exc;
}
catch(Error err) {
    error = err;
    throw err;
}
finally {
    ...
}
```

```
try {
    ...
} catch(RuntimeException exc) {
    error = exc;
    throw exc;
} catch(Error e) {
    error = e;
    throw e;
} finally {
    ...
}
```


Project.java

# braces



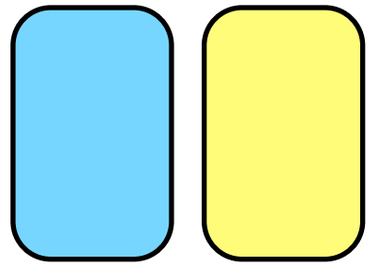
```
try {
    ...
}
catch(RuntimeException exc) {
    error = exc;
    throw exc;
}
catch(Error err) {
    error = err;
    throw err;
}
finally {
    ...
}
```

```
try {
    ...
} catch(RuntimeException exc) {
    error = exc;
    throw exc;
} catch(Error e) {
    error = e;
    throw e;
} finally {
    ...
}
```

# personal preferences?


Project.java

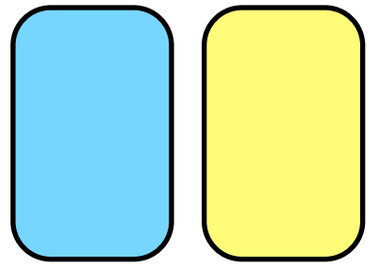
# for loop



```
public void run(TestResult result) {
    for (Test each : fTests) {
        if (result.shouldStop() )
            break;
        runTest(each, result);
    }
}

public void runTest(Test test, TestResult result) {
    test.run(result);
}
```

```
public void run(TestResult result) {
    for (Enumeration e= fTests.elements(); e.hasMoreElements(); ) {
        if (result.shouldStop() )
            break;
        Test test= (Test)e.nextElement();
        test.run(result);
    }
}
```

# for loop



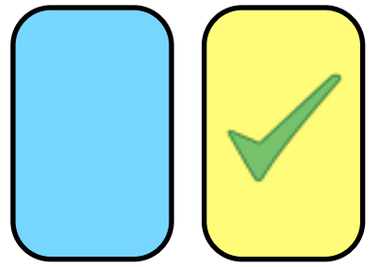
```
public void run(TestResult result) {
    for (Test each : fTests) {
        if (result.shouldStop() )
            break;
        runTest(each, result);
    }
}

public void runTest(Test test, TestResult result) {
    test.run(result);
}
```

```
public void run(TestResult result) {
    for (Enumeration e= fTests.elements(); e.hasMoreElements(); ) {
        if (result.shouldStop() )
            break;
        Test test= (Test)e.nextElement();
        test.run(result);
    }
}
```

JUnit

TestSuite.java

# for loop



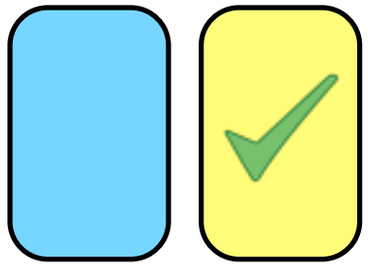
```
public void run(TestResult result) {
    for (Test each : fTests) {
        if (result.shouldStop() )
            break;
        runTest(each, result);
    }
}

public void runTest(Test test, TestResult result) {
    test.run(result);
}
```

```
public void run(TestResult result) {
    for (Enumeration e= fTests.elements(); e.hasMoreElements(); ) {
        if (result.shouldStop() )
            break;
        Test test= (Test)e.nextElement();
        test.run(result);
    }
}
```

JUnit

TestSuite.java

# for loop



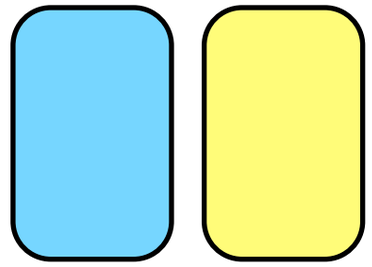
```
public void run(TestResult result) {
    for (Test each : fTests) {
        if (result.shouldStop() )
            break;
        runTest(each, result);
    }
}

public void runTest(Test test, TestResult result) {
    test.run(result);
}
```

```
public void run(TestResult result) {
    for (Enumeration e= fTests.elements(); e.hasMoreElements(); ) {
        if (result.shouldStop() )
            break;
        Test test= (Test)e.nextElement();
        test.run(result);
    }
}
```

language features?

JUnit

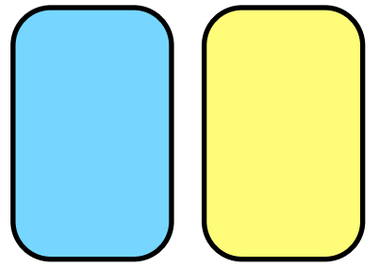
TestSuite.java



# Refactoring

```
boolean isTimedout = false;
if (timeout > 0) {
     isTimedout = delta > timeout;
}
```

```
boolean isTimedout = timeout > 0 && delta > timeout;
```

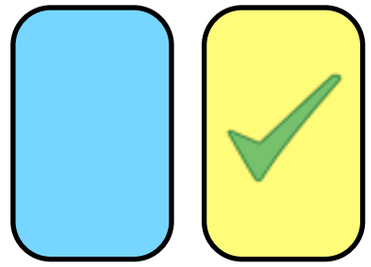


# Refactoring

```
boolean isTimedout = false;
if (timeout > 0) {
     isTimedout = delta > timeout;
}
```

```
boolean isTimedout = timeout > 0 && delta > timeout;
```

NioEndpoint.java

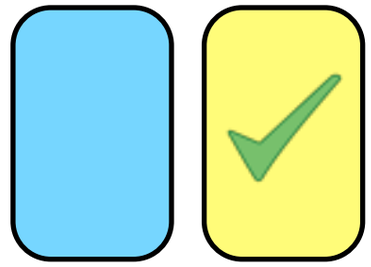


# Refactoring

```
boolean isTimedout = false;
if (timeout > 0) {
     isTimedout = delta > timeout;
}
```

```
boolean isTimedout = timeout > 0 && delta > timeout;
```

NioEndpoint.java



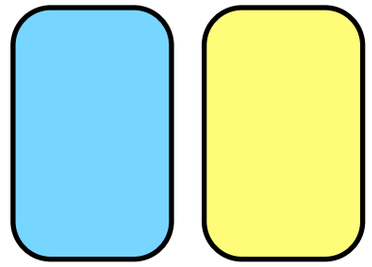
# Refactoring

```
boolean isTimedout = false;
if (timeout > 0) {
     isTimedout = delta > timeout;
}
```

```
boolean isTimedout = timeout > 0 && delta > timeout;
```

cleaner code?

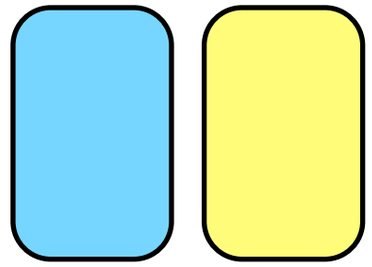
NioEndpoint.java



# dummy variables

```
try {
    in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

```
try {
    int x = in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

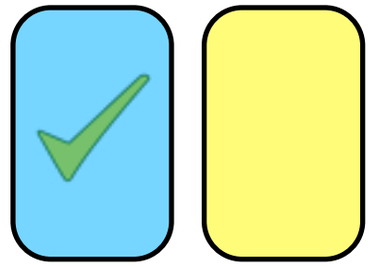


# dummy variables

```
try {
    in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

```
try {
    int x = in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

JSSESupport.java

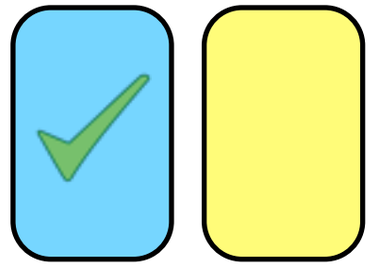


# dummy variables

```
try {
    in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

```
try {
    int x = in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

JSSESupport.java



# dummy variables

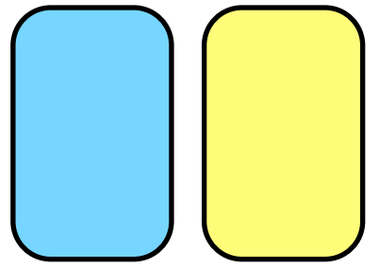
```
try {
    in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

```
try {
    int x = in.read(b);
} catch(SSLException sslex) {
    log.info("SSL Error getting client Certs",sslex);
    throw sslex;
} catch (IOException e) {
    // ignore - presumably the timeout
}
```

# to avoid warnings?

JSSESupport.java

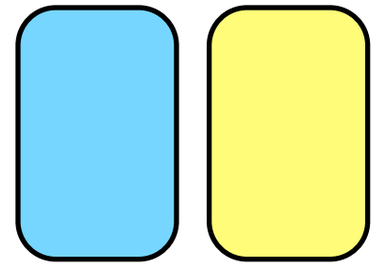
# serialVersionUID



```
public class AssertionFailedError extends AssertionError {

    private static final long serialVersionUID= 1L;

    public AssertionFailedError() {
    }

    public AssertionFailedError(String message) {
        super(message);
    }
}
```

```
public class AssertionFailedError extends Error {

    public AssertionFailedError () {
    }
    public AssertionFailedError (String message) {
        super (message);
    }
}
```

# serialVersionUID



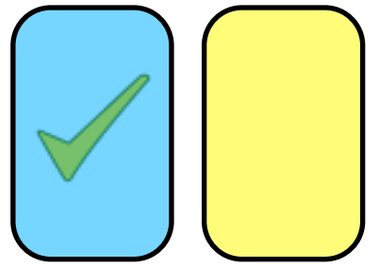
```
public class AssertionFailedError extends AssertionError {

    private static final long serialVersionUID= 1L;

    public AssertionFailedError() {
    }

    public AssertionFailedError(String message) {
        super(message);
    }
}
```

```
public class AssertionFailedError extends Error {

    public AssertionFailedError () {
    }
    public AssertionFailedError (String message) {
        super (message);
    }
}
```

JUnit

AssertionFailedError.java

# serialVersionUID



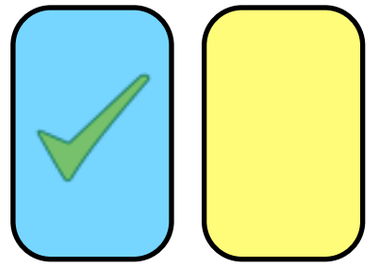
```
public class AssertionFailedError extends AssertionError {

    private static final long serialVersionUID= 1L;

    public AssertionFailedError() {
    }

    public AssertionFailedError(String message) {
        super(message);
    }
}
```

```
public class AssertionFailedError extends Error {

    public AssertionFailedError () {
    }
    public AssertionFailedError (String message) {
        super (message);
    }
}
```

JUnit

AssertionFailedError.java

# serialVersionUID



```
public class AssertionFailedError extends AssertionError {

    private static final long serialVersionUID= 1L;

    public AssertionFailedError() {
    }

    public AssertionFailedError(String message) {
        super(message);
    }
}
```

```
public class AssertionFailedError extends Error {

    public AssertionFailedError () {
    }
    public AssertionFailedError (String message) {
        super (message);
    }
}
```

tool support? eclipse?

JUnit

AssertionFailedError.java

other stuff that might influence a Java codebase:

- better understanding of exception handling
- old optimization patterns
- double checked locking
- how to synchronize code
- string building
- best practice has become worst practice
- new java libraries (eg collection and concurrency)
- nio
- fashion and trends (ejb, IoC, remoting)
- setup and initialization / configuration
- ... and much more

# About Software Development

Few software projects are like running on a paved road where you can see the ...


... goal in the end of the road.

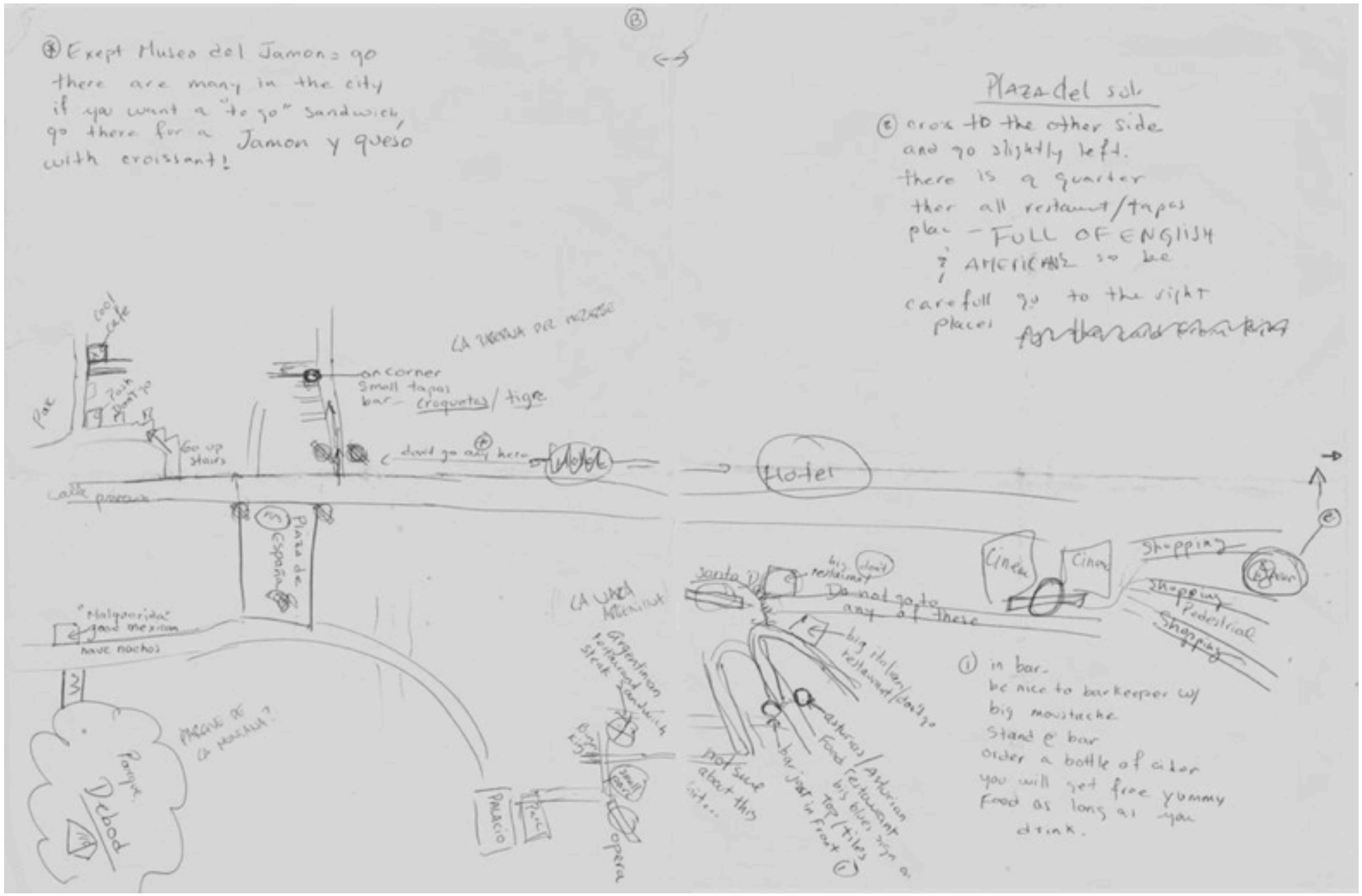
Most projects are more like...

extreme orienteering

in impossible terrain

with a group of people

in the dark




with only a sketchy map as guidance

You rush software developers, you get rotten miracles...

# You rush software developers, you get rotten miracles...

There are only two kinds of codebases: the ones people complain about and the ones nobody cares about anymore...

(inspired by a similar quote by Bjarne Stroustrup)