# NODE Technical Book Club

## A Philosophy of Software Design - John Ousterhout

# Better Together or Better Apart?

Should two functionalities be in the same module or separate modules?

- Smaller components:
  - Easier to understand subcomponents.
  - More interfaces to manage -> more complexity
  - Seperation -> more unknown unknowns
  - More duplication

Together is better if they are closely related.

Indicators:

- Shared information
- Being used together
- Conceptual overlap
- Hard to understand one without the other

# 🚩 Red Flag : Repetition

If the same piece of code appears over and over again, that's a red flag that you haven't found the right abstractions.

```
switch (common->opcode) {
  case DATA: {
    DataHeaders header = received->getStart<DataHeader>();
    if (header == NULL) {
      LOGCWARNING, "%s packet from %s too short (%u bytes)",
      opcodeSymbol (common->opcode) ,  received->sender->toString(), received->len);
      return;
    }
  }
  case GRANT: {
    GrantHeaders header = received->getStart<GrantHeader>();
    if (header == NULL) {
      LOGCWARNING, "%s packet from %s too short (%u bytes)", opcodeSymbol (common->opcode) , received->sender->toString(), received->len);
    return;
    }
  }
  case RESEND: {
    ResendHeaders header = received->getStart<ResendHeader>();
    if (header == NULL) {
      LOGCWARNING, "%s packet from %s too short (%u bytes)", opcodeSymbol (common->opcode) , received->sender->toString(), received->len);
    return;
    }
  }
}
```

# Repeated logging for each case

# Refactored to:

```
switch (common->opcode) {
  case DATA: {
    DataHeaders header = received->getStart<DataHeader>();
    if (header == NULL) goto packetTooShort;
  }
  case GRANT: {
    GrantHeaders header = received->getStart<GrantHeader>();
    if (header == NULL) goto packetTooShort;
  }
case RESEND: {
  ResendHeader* header = received->getStart<ResendHeader>();

if (header == NULL) goto packetTooShort;

}

packetTooShort:
LOG(WARNING, "%s packet from %s too short (%u bytes)", opcodeSymbol (common->opcode) , received->sender->toString(), received->len);

return;
```

# 🚩 Red Flag : Special-General Mixture

It's a red flag when a general-purpose mechanism also contains code specialized for a particular use of that mechanism.

This makes the mechanism more complicated and creates information leakage between the mechanism and the particular use case.

## Examples:

- Insertion cursor and selection -> better apart
- Separate logging methods for each case -> better together

# Splitting and Joining Methods

- Each method should do one thing and do it completely.
- Splitting up a method only makes sense if it results in cleaner abstractions.

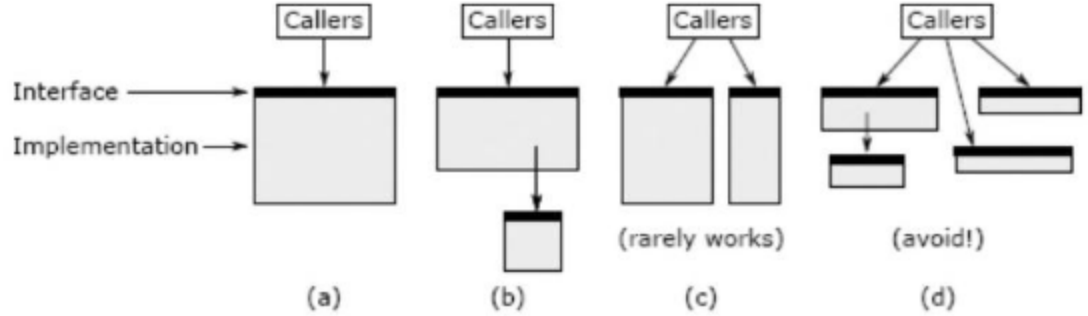


**Figure 9.3:** A method (a) can be split either by by extracting a subtask (b) or by dividing its functionality into two separate methods (c). A method should not be split if it results in shallow methods, as in (d).

- Join methods if:
  - It leads to deeper methods
  - It eliminates duplication
  - It eliminates unnecessary interfaces or data structures
  - It results in better encapsulation

# 🚩 Red Flag : Conjoined Methods

It should be possible to understand each method independently. If you can’t understand the implementation of one method without also understanding the implementation of another, that’s a red flag.

# A Different Opinion: Clean Code

- In the book Clean Code, Robert C. Martin argues that functions should be broken up based on length alone.

> The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

- BUT: Depth is more important than length. Don't sacrifice depth for length.

# Define Errors Out of Existence

- Exception handling is one of the worst sources of the complexity.
- AIM: To reduce number of places where exceptions must be handled

- An exception distrupts the normal flow of the code.
  - When an exception is caught, we need to either:
    - Try to fix it and complete the work.
    - Abort it and report upwards.

# Example: Unset in TCL

- It throws an exception if the variable doesn't exist.
- It should have simply returned without doing anything -> define errors out of existence

# Example: File Deletion in Windows

- In an OS, deleting an open file is tricky.

- Windows: returns an error if the file is open.
- Unix: deletes the file but it's still accessible until the last handle is closed -> define errors out of existence

# Example: Java Substring Method

- Java's substring method throws an exception if the index is out of bounds.
    - It should have returned an empty string.

# Mask Exceptions

- Handle exceptional conditions at a low level in the system so that they don't propagate upwards.
- Example: TCP sends lost packets again without informing the application.
- Example: NFS client retries requests over and over again if the server fails to respond.

# Exception Aggregation

- Handle multiple exceptions in one place.
- Example: Missing parameters in a Web server.

# From:

Dispatcher:

```
...
if (...) {
  handleUrl1(...);
} else if (...) {
  handleUrl2(...);
} else if (...) {
  handleUrl3(...);
} else if (...)
  ...
}
...
```

handleUrl1:

```
...
try {
  ... getParameter("photo_id")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
try {
  ... getParameter("message")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
```

handleUrl2:

```
...
try {
  ... getParameter("user_id")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
```

handleUrl3:

```
...
try {
  ... getParameter("login")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
try {
  ... getParameter("password")
  ...
} catch (NoSuchParameter e) {
  ...
}
...
```

**Figure 10.1:** The code at the top dispatches to one of several methods in a Web server, each of which handles a particular URL. Each of those methods (bottom) uses parameters from the incoming HTTP request. In this figure, there is a separate exception handler for each call to getParameter; this results in duplicated code.

# To:

Dispatcher:

```
...
try {
  if (...) {
    handleUrl1(...);
  } else if (...) {
    handleUrl2(...);
  } else if (...) {
    handleUrl3(...);
  } else if (...)
    ...
  }
} catch (NoSuchParameter e) {
  send error response;
}
...
```

handleUrl1:

```
... getParameter("photo_id")
... getParameter("message")
...
```

handleUrl2:

```
... getParameter("user_id")
...
```

handleUrl3:

```
... getParameter("login")
... getParameter("password")
...
```

**Figure 10.2:** This code is functionally equivalent to Figure 10.1, but exception handling has been aggregated: a single exception handler in the dispatcher catches all of the NoSuchParameter exceptions from all of the URL-specific methods.

# Just Crash

- If an error is difficult or impossible to handle and it's unlikely to occur, just crash.
- Example: Out of memory
- Counterexample: malloc in C returns NULL if out of memory.

# Design it Twice

- It's unlikely that you will get the design right the first time.
- Consider multiple options for the major design decisions.
- Try to pick approaches that are radically different from each other.
- Compare them and finalize the design.

## Questions to ask:

- Is one alternative easier to use?
- Does one alternative have a simpler interface?
- Is one interface more general-purpose?
- Does one interface enable a more efficient implementation?

# Comments