1

Compare the `write_message` methods of the console-based and file-based backends. There is hidden coupling between them. What is the hidden coupling? What changes does this hidden coupling inhibit? How would you refactor them to eliminate it.

Answer Assuming you want e-mails to be sent consistently, the ('-' * 79) separator is shared between them. In fact, the code between them is the same, but acts on different encodings (byte strings vs. ASCII strings). A solution is to unify both implementations of `write_message` into the superclass (console backend), but abstract over the encoding. Can put this in send_messages, or move part of `write_message` to the superclass (console backend), or a shared `format_email` method. There are also many other options.

Example code:

File-based

```
    def write_message(self, message):
        msg = message.message().as_bytes();
        super().formatted_write(msg)
```

Superclass

```
    def formatted_write(self, message):
        self.stream.write('%s\n' % message)
        self.stream.write('-' * 79)
        self.stream.write('\n')
```

2

The send_messages methods of both the SMTP-based and console-based (and file-based, by inheritance) backends have a common notion of failing silently according to an option.

2.a

What is the essence of the "fail-silently" pattern? In other words, if I gave code where all identifiers were obfuscated, how would you identify code that implemented the "fail-silently" feature? Give your answer either as a very specific description of data- and control-flow to look for, or as a code skeleton.

Answer The answer we are looking for is that "failing silently" is defined by wrapping code in a try/catch handler that conditionally ignores exceptions. But let's take a step back to explain why we can confidently say this is the right answer, rather than just one perspective.

We first inspect the idea of failure in programming, and present a conceptual analysis using the techniques of Daniel Jackson's *Essence of Software*. Failure is not a standalone

concept, but is actually an offshoot of the idea of an atomic action. An atomic action has some view of the world, some way to inspect the world's state and see a change before and after it executes, and an ability to perform the action. An action can be invoked multiple times; invoking it twice on the same state of the world should produce the same results.

Now, "failure" is a possible result of performing the action. A clean and desirable property of failure: if the action reports failure, then the relevant parts of the world state are the same as they were before the failed attempt. Many instantiations of the failure concept also come with a reason; a reason is meaningful if attempting to perform the action again without changing the reason for failure results in another failure, and removing the failure reason results in the action succeeding upon retry, or at least failing for a different reason.

"Failing silently" can be defined at the conceptual level based on the ideas of action and failure: it means that if an action fails, the state will not change, but it will not report failure in its return value. A corollary: while there are many ways in general to compose multiple actions into one larger action, if all failures are silent, then performing the compound action should execute each constituent action unconditionally.

Now, back to the concrete:

In this setting, an action is a method call and failure comes in the form of an exception. It follows immediately that making an action fail silently means catching and suppressing its exceptions, and having a setting for failing silently means some conditional that determines whether such an exception is suppressed or propagated.

Here's a code fragment for the idea of failing silently. There actually cannot be too many variants of this pattern: the body of the action must be within a `try` block, and whether an exception raised by the body is propagated onwards (i.e.: re-raised) must be determined by the value of the `fail_silently` variable (i.e.: control-dependent on it, and therefore within some kind of conditional based on it).

```
try:
  ...
except (kinds of exception) as e:
  if not self.fail_silently:
    raise e
```

2.b

What are the design decisions which are the same between the two backend's implementation of fail-silently, and how might a change to these decisions affect both implementations? Think about other policies for how the application should handle exceptions other than "fail at the top-level immediately for all exceptions" and "silently drop all exceptions."

Answer Taking as a given that failure comes in the form of exceptions, the design decisions affecting both backends can be described as follows:

During the process of sending messages (open connection, send messages, close connection), the current design allows for exactly two ways to deal with errors:

- Any error encountered will make the process of sending messages abort right when the error happens **and** the procedure will report the error.
- All errors will be ignored; when an error occurs, the application will continue to try sending the messages **and** the errors will be completely suppressed (not logged).

Similar to question 1 of this case study, if there's a change to the decision of what error handling policies should exist, both backend implementations will have to change.

Some examples of other policies would be:

- Always fail in case of specific errors.
- Always ignore certain errors.
- Retry operation when specific errors happen.

2.c

Sketch how to refactor the code to eliminate this hidden coupling. A successful solution should give code for "failing silently" that can be used in contexts unrelated to e-mail. (Hint: Use Python's with statement)

Answer When you see two blocks of similar code with identical snippets on the inside, that can be refactored out into a function. But in cases like this, where you see two blocks of similar code with identical snippets on the outside, that calls for a higher-order function.

For a family of such templates (of the form "do something, then do the body, then do something else when the body is finished or has raised an exception"), Python has special syntax for doing this: you can use a with-statement instead of calling a higher-order function.

```
with error_handling_policy():
  self.connection.sendmail(from_email, recipients,
   ↪  message.as_bytes(linesep='\r\n'))
```

The return type of `error_handling_policy()` is a Python context manager, i.e.: something with `__enter__` and `__exit__` methods.

The equivalent Java code would be this:

```
this.errorHandlingPolicy.doFailableAction(() -> {
  this.connection.sendmail(...);
});
```

where `doFailableAction` has many possible implementations.

Below is one example of a possible refactor for this exercise using Python.

Create two classes describing different error handling policies, and create an instance of one of those classes in BaseEmailBackend's constructor.

```
  class FailSilently:
    def __enter__(self):
      pass
```

```
    def __exit__(self, type, value, traceback):
      return True

  class PropagateErrors:
    def __enter__(self):
      pass
    def __exit__(self, type, value, traceback):
      return False

  class BaseEmailBackend:
    def __init__(self, fail_control=FailSilently(), **kwargs):
          self.fail_control = fail_control
```

The next step is to refactor the existing code that implements the fail_silently concept to use the instance of FailSilently. Example of refactoring for one of the methods.

```
# refactor of method _send on the SMTP backend
def _send(self, email_message):
  """A helper method that does the actual sending."""
  if not email_message.recipients():
    return False
  encoding = email_message.encoding or settings.DEFAULT_CHARSET
  from_email = sanitize_address(email_message.from_email, encoding)
  recipients = [sanitize_address(addr, encoding) for addr in
 ↪  email_message.recipients()]
  message = email_message.message()
  with self.fail_control:
    self.connection.sendmail(from_email, recipients,
     ↪  message.as_bytes(linesep='\r\n'))
    return True
  return False
```

There is also a less elementary implementation of FailSilently using Python's `contextlib.contextmanager` .

```
@contextmanager
def FailSilently():
  try:
    yield
  except:
    pass
```

A very technical sidenote: The process of identifying and extracting out common parts of multiple code snippets is called *anti-unification*. When the code snippets in question have scaffolding in common surrounding an inner body that differs, this is called a *second-order anti-unification*.

It might be tempting to let `FailSilently` take a boolean to indicate if it should be silent or not, to be a bit more DRY.

```
class FailSilently:
  def __init__(self, fail_silently = true):
```

```
    self.fail_silently = fail_silently
  def __enter__(self):
    pass
  def __exit__(self, type, value, traceback):
    return self.fail_silently
```

This however is worse for multiple reasons: not only does this code suffers from boolean blindness but it also violates https://www.linguistic-antipatterns.com/?tab=%22Inappropriately-specific-name%22

For the boolean blindness issue, the traditional solution is to replace the boolean with an enum. But if one adopts that solution, what would the enum be? It highlights the issue in this design. And that at the very least the function name should be less specific and the argument more specific.

The suggested solution treats the different error-handling policies as different members in the open sum of `ErrorPolices`. The set of possible `ErrorPolicies` can be expanded and contracted at will.

3

The `__init__` method of the file-based backend is complicated because of the impedance mismatch between the file path argument it accepts and the actual requirements on files.

3.a

What are the concrete restrictions it is placing on file paths? What are the underlying design decisions behind these restrictions?

Answer Concrete restrictions:

- File path must be a string
- If file path points to an existing resource it must be a directory
- The directory pointed by the file path must be writable

Examples of design decisions that contributed to the origin of code that checks the above restrictions:

- Emails written by the file backend will be written to a directory
- Users of the file backend should be able to specify which directory the e-mails should be written to.

3.b

What changes to the system's overall design or assumptions may change this code?

Answer Most changes that would somehow impact the design decisions mention in 3.1 would cause the code to change. For example:

- A decision that all e-mails should be written to a single .txt file.
- A decision that e-mails will be stored in N different directories depending on their content.

Note how in the code there are a lot of assumptions about how permissive the OS in which the code will run is. If the server starts running on an OS with a more restrictive permissions system, the code will have to change.

3.c

Sketch how to refactor this method to embed the design decisions identified in 3.a directly into the code. Hint: Your answer should change the API of init.

Answer The solution is to replace the `file_path` with an instance of a new class called `ExistentWritableDirectory`. An instance of this class would perform all the validations that are currently performed on the file-based backend constructor.

```
class ExistentWritableDirectory:
  def __init__ (self, file_path):
    self.file_path = os.path.abspath(file_path)

    try:
        os.makedirs(self.file_path, exist_ok=True)
    except FileExistsError:
        raise ImproperlyConfigured(
            'Path exists, but is not a directory: %s' % self.file_path
        )
    except OSError as err:
        raise ImproperlyConfigured(
            'Could not create directory: %s (%s)' % (self.file_path,
             ↪ err)
        )
    if not os.access(self.file_path, os.W_OK):
        raise ImproperlyConfigured('Could not write to directory: %s'
         ↪ % self.file_path)

  def get_path(self):
    return self.file_path
```

Refactor the filebased backend to use `ExistentWritableDirectory`

```
def __init__(self, *args,
 ↪ destination_directory=setting.get_email_directory(), **kwargs):
 self._fname = None
 self.destination_directory = destination_directory
 # Finally, call super().
 # Since we're using the console-based backend as a base,
```

```
 # force the stream to be None, so we don't default to stdout
 kwargs['stream'] = None
   super().__init__(*args, **kwargs)

def _get_filename(self):
 """Return a unique file name."""
 if self._fname is None:
   timestamp = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
   fname = "%s-%s.log" % (timestamp, abs(id(self)))
   self._fname = os.path.join(self.destination_directory.get_path(),
    ↪  fname)
 return self._fname
```

Note that before the refactoring, a programmer could only rely on the fact that the given file path was an existing and writable directory in the code that runs after all the checks in `__init__` and in the methods of the backend. By moving those checks to `ExistentWritableDirectory` those facts become invariants of the class and can be used in many places aside from just the backend. And furthermore, if the directory is generated not from a string but from somewhere else, e.g.: a routine that creates a new writeable directory, then these checks may not need to be performed at all.

The refactor also makes explicit the design decision of allowing users of file backend to specify the target directory. Also, note how it encapsulates assumptions about the server's OS. If those assumptions change, there's no longer a risk of introducing new bugs in the file backend while adjusting code related to directories.

A recap of benefits:

- It makes a useful concept explicit -- As a consequence of this enriched type, the sought properties in a file path can be "remembered"
- It directly places requirements into the function signature (avoiding an error of modular reasoning)
- It makes the constructor harder to misuse (in service of the representable/valid principle)

4

Look at the `__init__` and message methods of `EmailMessage` . Notice the different representations and handling of different e-mail headers such as from, bcc, to.

4.a

Consider the subject, date, and Message-ID headers. Find a way that summarizes how all three of them are handled using at most 15 words. (Our reference solution uses 9 words.)

Answer Use the given value, else use the default value.

4.b

Find 2 parameters that are treated similarly to the `to` parameter, and summarize how they are handled.

Answer ”cc” and ”Reply-to”. Both of them are used from the header if present, both should be a list or a tuple that then is concatenated, and if they’re not a list or a tuple an exception will be raised.

4.c

Explain what code in Django would need to change to support each of the following features

4.c.i The ability to, at the end of each day, collect the list of all e-mail addresses that had been messaged that day.

Answer

Solution 1 Write down a list of all headers that contain e-mail addresses, check for them in both the arguments and the extra_headers field

Solution 2 Create a data structure to store the sent e-mails, pass it to init, and change the send method to store the e-mails addresses once the send succeeds.

4.c.ii The ability to set a policy that certain headers are required, e.g.: set Alternate-Recipient or Content-Language as mandatory headers.

Answer Depending on the header, it’s necessary to add checks in the different ways the parameter can be passed. If the header comes as a parameter but can also be part of the headers parameter, then a check that verifies the parameter and the headers need to add. If it can only be part of headers, then add the check to the headers parameter.

4.c.iii The SMTP spec is expanded, and some e-mail addresses are given a globally-unique integer ID, and addresses in headers can be specified by this ID rather than by name. After a few years, it becomes routine for people to write ”to:12345 cc:bob@example.com, 67891011, dave@example.com” instead of ”to:alice@example.com cc:bob@example.com,carol@example.com,dave@example.com”. Assume the underlying libraries have been upgraded accordingly. Hint: There is some code in `__init__` that, while arguably not wrong if this change occurs, would have been written differently if SMTP was like this from the start.

Answer `__init__` on lines 208-230 of the given revision contains a series of checks like the following:

```
if to:
  if isinstance(to, str):
    raise TypeError('"to" argument must be a list or tuple')
  self.to = list(to)
else:
  self.to = []
```

Each of these `isinstance` checks was coded from the assumption that the primary/only type error for these fields is passing in a string. With this change, an equally feasible mistake is to pass in an int. The check will need to be updated accordingly to also guard against ints.

Note: As of Python 3.9.1, this code would happen to give an error anyway ("TypeError: 'int' object is not iterable"). However, this would not be sufficient (i.e.: were one to write code from scratch that calls the `list` function on an argument with the idea that passing in an int for that argument is a mistake, they would likely not choose to do it this way). Reasons are

1. this is not a very informative error message
2. this error message is an implementation detail in that it gives information about the implementation of `__init__` , not information about the way `__init__` is called, and, most importantly
3. it is generally not a stable guarantee that a type does not implement an interface, and thus the correctness of this code should not rely on no-one implementing a future Python change that would make the `list(to)` call succeed. In this case, there actually was a formal proposal that would have made this line succeed but behave undesirably. (https://www.python.org/dev/peps/pep-0276/) The proposal was rejected for sound reason, though there is no true guarantee it will never be done in the future. In broader strokes, belief in code's robustness and future robustness should not rest on complex arguments and external factors like that in the preceding few sentences.

4.d

Explain the design of the abstract concept of different kinds of headers, i.e.: how would you explain what headers are, what variations there are, and how they work to someone who had never seen them before? How are these ideas expressed in the code? How does this make the code complicated?

Answer Headers are designed to convey information to the system handling the e-mail. There can be different types (e.g. list vs string), and they can be mandatory or not. Ultimately they all become a mapping of string key to string value inside the message.

For a detailed explanation of the abstract concept of headers give a quick scan to the Internet Message Format (RFC). The description of headers starts on page 7, and after that, there are sections dedicated to each kind of header (E.g Destination address fields, identification fields, ...)

The code expresses these as primitive types and identifies them either as a passed in argument to the constructor, or the name of the key. This requires a lot of special casing for different types (e.g. the repeated list/tuple checks for to, cc, bcc, etc.). Further some of the headers are kept as k/v pairs inside `self.extra_headers` whereas others are stored as properties on the `EmailMessage` itself. Both of these complicate the code because you need to do a lot of reflection to see what type you have, as well as look for properties in multiple places, knowing which can exist on the message itself and which cannot, which are required and which are not, etc..

Answer of Mitchell Rivet: From wikipedia: ”Each message has exactly one header, which is structured into fields. Each field has a name and a value”. There is a separator (”:”) between header and value. ”From” and ”Date” are required for a valid header. These basic formatting rules are expressed in code line by line, and are very much ”dark knowledge”.

If I know nothing about e-mail headers, these implementation requirements are not clear to me (or at least quickly accessible) through this code. It would be much quicker to grasp if we abstracted these.

4.e

Sketch how to refactor this code based on the abstract design of headers. Your answer may change the API.

Answer Two majors families of solutions are to have a type for types of e-mail headers, or to have a type for e-mail headers. In the former, the `EmailMessage` class maintains a list of ”header schema descriptions,” capable of inspecting an unstructured key/value list and extracting relevant header information. In the latter, the caller is responsible for passing in headers as structured data.

Here is an example from Jimmy Koppel implementing both.

https://gitlab.com/-/snippets/2274852

In summary, a submission’s refactoring should contain the following:

1. An explicit encoding of the mapping between the header name and its type (EmailAddress , DateTime etc).
2. Default values for each header should be explicit and in one place.
3. Validation of the required headers for an email should be baked in.

4.f

Repeat question 3 for this new design. These questions are answered using this refactored version of the code https://gitlab.com/-/snippets/2274852

Explain what code in Django would need to change to support each of the following features

4.f.i The ability to collect a list of all e-mail addresses to which e-mails were sent each day.

Solution 1 Check for headers whose type in the schema is `EmailAddress` or `EmailAddressList` where the header name is not equal to 'From'.

Solution 2 No change in comparison to how it was before the refactoring.

4.f.ii The ability to set a policy that certain headers are required, e.g.: set Alternate-Recipient or Content-Language as mandatory headers.

Answer Set `HeaderRequired` in the header schema for the targeted header.

4.f.iii The SMTP spec is expanded, and some e-mail addresses are given a globally-unique integer ID, and addresses in headers can be specified by this ID rather than by name. After a few years, it becomes routine for people to write "to:12345 cc:bob@example.com, 67891011, dave@example.com" instead of "to:alice@example.com cc:bob@example.com,carol@example.com,dave@example.com". Assume the underlying libraries have been upgraded accordingly. Hint: There is some code in `__init__` that, while arguably not wrong if this change occurs, would have been written differently if SMTP was like this from the start.

Answer Change the validate method of the `EmailAddress` class.

5

Find one more instance of a design-level concept which is only indirectly expressed in this code, and explain how to refactor it. Code smells that may help you find examples include: if-statements, the use of base types such as bool or string to represent more complicated concepts, and methods that have multiple return types.

Answer One example: e-mail attachments. As one student writes:

```
The design concept of an email attachment is not well-embedded in this code,
and consequently there are many rough edges where the code that does handle it deals w
takes instances of attachment representations or primitive types it builds into attac
Many methods are dealing with far too many different input types. This is exemplified

> Convert the content, mimetype pair into a MIME attachment object.
If the mimetype is message/rfc822, content may be an email.Message or EmailMessage ob

The central interface everything is operating on is a the `EmailMessage`'s .attach me
content and mimetype, and itself will handle a lot of variants, ultimately just doing
```

```
# If MIMEBase: no content/no mimetype
self.attachments.append(filename)
# Otherwise
self.attachments.append((filename, content, mimetype))
```

I would therefore refactor this by expressing the fact that there are multiple types of attachments, and they can either have a filename and no content or mimetype, or they can have a filename, content and mimetype. Thus we can create an attachment type, and then having various factory methods to create them by passing in filename and optionally (content + mimetype). This way you wouldn't have methods that are force to handle all of the optionals, rather you would create specific types of `Attachment` s based on what you were intending to provide.

We could represent mimetype as its own ADT as well, and have methods on it to `guess_type` etc., from the filename, so that our `Attachment` class is always provided with one if it exists.

More examples: `forbid_multi_line_headers` (no explicit concept of header parsing or valid headers), HTML vs. plaintext e-mails

6

Bonus (optional): There are also at least two violations of the representable/valid principle (next unit's lesson) in this code (at least one where it is possible to represent invalid states, and at least one where there are multiple states of `EmailMessage` that correspond to the same e-mail). Find them. How would you eliminate them? How would this simplify the code?

Answer Instance 1:

```
    if self.use_ssl and self.use_tls:
        raise ValueError(
            "EMAIL_USE_TLS/EMAIL_USE_SSL are mutually exclusive, so
             ↪ only set "
            "one of those settings to True.")
```

(smtp.py, line 31)

Instance 2:

```
Some headers can be specified both as a property of the `EmailMessage` object, and in
E.g.: the "from" field. Additionally, it is possible to add multiple copies of the sa
because e-mail headers are case insensitive, but the extra_headers map is case-sensi
```

Related:

```
When a program passes one of the `To`, `Cc` or `Reply-To` headers in the `extra_header
but does not pass a value in the `to`, `cc`, or `reply_to` parameters in the construct
this would cause the `_set_list_header_if_not_empty` method to leave these headers e
were passed the constructor.  This is related to the "no fluff/confusion" part of the
```