#readings #cleancode #topublish

Clean Code - A Handbook of Agile Software Craftsmanship

Notes

There is a bit of a contrast between this book and *Philosophy of Software Design* regarding

1. how methods (and classes) should be factored. The latter suggests that a method should be responsible for all relevant operations, even when long, rather than splitting into several methods that do not hold independently. This is, of course, a matter of balance.
2. how comments should be handled. The latter is much more benign and sees these as a useful (and necessary) tool of software design, while this book sees them (IMO) as a necessary evil.

Chapter 1: Clean Code

In this book you will find sporadic references to various principles of design. These include the Single Responsibility Principle (SRP) (see Chapter 10), the Open Closed Principle (OCP), and the Dependency Inversion Principle (DIP) (see also Chapter 10) among others. These principles are described in depth in Agile Software Development, Principles, Patterns, and Practices.

Open-Closed principle: Classes should be open for extension but closed for modification.

Chapter 2: Meaningful Names

Main points

1. **Use intention-revealing names**: avoid for example, short variable names unless the scope of the variable is small and make sure that the name exposes what the variable is doing;
2. **Avoid disinformation**: using similar spelling for similar concepts is information, using inconsistent spelling is misinformation;
3. **Make meaningful distinctions**: noise words are redundant. The word variable should never appear in a variable name and the word table should never appear in a table name. This rule suggests that one should avoid using, for example, `NameString`, as it is obvious that `Name` should have type `string`;
4. **Use pronounceable names**;
5. **Use searchable names**: single-letter names should only be used as local variables inside short methods. *The length of a name should correspond to the size of its scope*;
6. **Avoid encodings**;
7. **Avoid mental mapping**;
8. **Class names**: classes and objects should have noun or noun phrase names;

9. **Method names**: method names should have verb or verb phrase names. Accessors, mutators and predicates should be name for their value and prefixed with `get`, `set` and `is`. When constructors are overloaded, use static factory names;
10. **Do not be cute**: mean what you say and say what you mean;
11. **Pick one word per concept**: the function names have to stand alone, and they have to be consistent in order for you to pick the correct method without any additional explanation;
12. **Use solution domain names**;
13. **Use problem domain names**;
14. **Add meaningful context**: this section has a good example of refactoring. Rather than having the complexity at the level of the method `printGuessStatistics` , this is split into several internal methods of the class `GuessStatistics`;
15. **Do not add gratuitous context**;

Chapter 3: Functions

There are several powerful tools at your disposal for making things clearer:

1. Method extractions - Rather than having a method doing several things (checking if a condition is true, performing a computation if said condition is `True`, …), make it do only one thing;
2. Renaming;
3. Reestructuring;

Note that, however, *doing one thing* is rather subjective and this would seem to conflict with *Philosophy of Software Design*, where it is suggested that a method should be responsible for everything, rather than splitting into several methods that do not hold independently.

Small

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

Blocks within `if/else/while` statements should be one line long, which should likely be a function call. This both keeps the enclosing function small, but also adds documentary value because the function called within the block can have a nice descriptive name.

This also implies that functions should not be long enough to have nested structures: the level of a function should not be greater than one or two.

Functions should only do one thing - To be more precise: if the function is only doing the steps that are *one level below the stated name of the function*, then it is doing one thing.

```
The reason why we write functions is to decompose a larger concept
(in other words, the name of the function) into a set of steps at
the next level of abstraction.
```

Avoid using different levels of abstraction inside the same function.

```
Another way to know that a function is doing more than one thing
is if you can extract another function from it with a name that
is not merely a restatement of its implementation.
```

One level of abstraction per function

In order to make sure our functions are doing “one thing”, make sure that the statements within the function are all at the same level of abstraction.

Reading code from top to bottom: the stepdown rule

Every function should be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions.

- **TO** include circle creation, then random splitting and then random coloring;
  - **TO** include circle creation, we define a circle object and its properties (radius);
    - **TO** define a circle object, ….
  - **TO** include random splitting, we define the splitting points inside the circle, we connect these with a line segment and connect the line segment to the perimeter of the circle;
    - **TO** define the splitting points inside the circle…

Example:

```python
def generate_random_circle_image(...):
    circle = create_circle(...)
    splitting = create_random_splitting(circle, ...)
    coloring = create_random_coloring(coloring, ...)
    # other operations
    ...

def create_circle():
    Circle(x=...,y=...)
    # other operations
    ...

def create_random_splitting(circle, ...):
    splitting_points = create_splitting_points(circle, ...)
    line_segment = create_line_segment(splitting_points)
    #other operations
    ...

def create_splitting_points(circle, ...):
    ...
```

Note that, for each initial layer, there is a sublayer which gets explored. One would only move to the next step of the initial layer once all sublayers of the current step are explored. Each function introduces the next, and each function remains at a consistent level of abstraction.

Use descriptive names

Code is clean when each routine turns out to be pretty much what you expected. Half the battle to achieving that principle is choosing good names for small functions that do one thing. *The smaller and more focused a function is, the easier it is to choose a descriptive name.* Oftentimes, a good name avoids the introduction of a comment.

Use the same phrases, nouns and verbs in the function names you choose for your modules.

Function arguments

The ideal number of arguments for a function is zero (niladic), followed by one (monadic) and two (dyadic). Three arguments (triadic) should be avoid where possible and more than three arguments should not be used.

Because arguments take a lot of conceptual power, these should be removed as much as possible. As an alternative, use instance variables rather than arguments: an instance variable is a variable which is declared in a class but outside of the constructors, see example below. *Furthermore, arguments should be present at the same level of abstraction.*

```
class Car:
    color = "red"

    def park():
        ...
```

Common monadic forms

There are two common reasons to pass an argument into a function:

1. One is *asking a question* about the argument (does it exist, for example);
2. One is *operating on the input* argument, transforming it into something else and returning it;

A somewhat less common form of a single argument is an *event* (for example, number of retries). One should, however, be careful and clear when using this.

Try to avoid any monadic functions that do not follow these forms. One example is using an output argument instead of a return value: if a function is going to transform its input argument, the transformation should appear as the return value.

Flag arguments

Passing a boolean into a function is a bad practice because it immediately complicates the signature of the method, which has to do two things (one if `True` and another if `False`) . If possible, split the function into two.

Dyadic functions

To avoid dyadic function (especially when the two arguments do not have a natural cohesion), you might try to:

1. Include a method into a class so that you can use, for example, `outputStream.writeField(name)` rather than `writeField(outputStream, name)`;
2. Or make `outputStream` a member variable of the current class so that you do not have not pass it (i.e, when `writeField` is called it already knows which `outputStream` to use);
3. Or extract a new class `FieldWriter` that takes the `outputStream` in its constructor and has a `write` method.

Triads

Triads are much harder to understand than dyads and therefore should be avoided at all costs.

Argument objects

When a function seems to need more than two or three arguments, it is likely that *some of those arguments ought to be wrapped into a class of their own.*

Reducing the number of arguments by creating objects out of them is not cheating! *When groups of variables are passed together, they are likely part of a concept which deserves a name of its own.*

Example: `Circle(x: float, y: float, radius: float) -> Circle(center: Point, radius: float)`

Verbs and Keywords

In the case of the monad, the function and argument should form a very nice *verb/noun* pair. For example `write(name)` is evocative, because it informs the user that `name` (whatever that is) is being written. An even better name might be `writeField(name)`.

This (`writeField`) is an example of the *keyword* form of a function name. Using this form, one encodes the names of the arguments into the function name. For example, `assertEquals` might be clearer written as `assertExpectedEqualsActual(expected,equal)`, because this mitigates the problem of having to remember the order of the arguments.

Have no side effects

These are damaging mistruths that often result in strange temporal couplings and order dependencies. A caller who believes what the name of the function `checkPassword` says could erase the existing session data when she tries to check the validity of the user. This creates a temporal coupling, because `checkPassword` can only be called at certain times (i.e, when it is safe to initialize the session). Calling it out of order leads to session data being lost.

If there is a temporal coupling, make it explicit in the function's name. But again, guarantee that this does not violate the "Do one thing" principle…

Output arguments

In general, output arguments should be avoided. If the function has to change the state of something, then change the state of the owning object. This can be achieved by transforming the function into a class method.

Command query separation

Functions should either do OR answer something (see section on common monadic forms), but not both!

Extract try/catch blocks

These are confusing by themselves because they are doing two things:

1. Error processing;
2. Normal functioning; Therefore, it is better to extract the bodies of the `try` and `catch` blocks into functions of their own.

Error handling is one thing

A function that handles errors should do nothing else because functions are only meant to do one thing. If `try` exists in a function, then it should be the first thing to appear and there should be nothing afterwards.

Structured Programming

Dijkstra (on structured programming) said that every function, and every block within a function, should have one entry and one exit.

Following these rules means that there should be:

- Only one `return` statement;
- No `break` or `continue` statements in a loop;

Chapter 4: Comments

Comments are a necessary evil. We should write code so that its purpose is obvious, making comments superfluous. *The proper use of comments is to compensate for our failure to express ourselves in code.* Not only do comments display our failure to clarify something in code, which is already a sign that something could be improved, but they are also problematic to maintain, as comments often do not follow the evolution of code.

Truth can only be found in one place: the code.

Explain yourself in code

Example:

```
# check to see if the employee is eligible for full benefits
if (employee.flags & HOURLY_FLAG) && employee.age > 65:
```

is far inferior to `if employee.isEligibleForFullBenefits`.

Good comments

These are essentially:

1. Legal;
2. Informative;
3. An explanation of intent;
4. Clarification. *But before you do this one, take care that there is no better way and make sure that these are accurate*;
5. Warning of consequences. *For tests this can be replaced by a marker*;
6. Amplification. *If you really want to highlight something*;

Bad comments

Usually they are crutches or excuses for poor code or justifications for insufficient decisions, *amounting to little more than the programmer talking to himself.* These are essentially:

1. Mumbling. *Any comment that forces one to look in another module for the meaning of that comment has failed to communicate clearly and it is not worth it*;
2. Redundant comments. *Let code do the talking. Do not add docstrings just because*;
3. Misleading comments;
4. Mandated;
5. Journal (the modules should not be a log-book);
6. Noise (do not comment obvious stuff. If a comment is really necessary, than likely the code is not clear enough);
7. *Do not use a comment when one can use a function or a variable*;
8. Position markers;
9. Nonlocal information;
10. TMI;

Chapter 5: Formatting

Code formatting is about communication and communication is the professional developer's first order of business.

The functionality that you create today has a good chance of changing in the next release, but the readability of your code will have a profound effect on all the changes that will ever be made.

Structure the source file to be like a newspaper article:

- The name should be simple but explanatory;
- The topmost part of the file should provide the high-level concepts and algorithms;
- Detail should increase as we move downward;

Concepts that are closely related should be kept vertically close to each other.

Variable declarations

Should occur as close to their usage as possible.

Instance variables

On the other hand, should be declared at the top of the class. This should not increase the vertical distance of these variables, because in a well-designed class, they are used by many methods of the class.

Dependent functions

If one function calls another, they should be vertically close and the caller should be above the callee, if at all possible. This gives the program a natural flow.

Conceptual affinity

The stronger the affinity, the less vertical distance should be between them. Affinity can come through a direct dependency, but it can also be caused by a group of functions that perform a similar operation.

Indentation

A source file is a hierarchy rather than an outline. There is information which pertains:

1. the file as a whole,
2. to the individual classes within the file,
3. to the methods within the classes,
4. to the blocks within the methods,
5. to blocks with blocks

Each level of this hierarchy is a scope into which names can be declared and in which declarations and executable statements can be interpreted.

Chapter 6: Objects and Data Structures

Data Abstraction

A class does not simply push its variables out through getters and setters. Rather, it exposes abstract interfaces that allow the users to manipulate the essence of the data, without having to know the implementation. This is further clarified with Listings 6-3 and 6-4.

To reiterate, one should not expose the details of the data. Rather, one should express our data in abstract terms, but this is not achieved by simply using interfaces and/or getters and setters.

Data/Object Anti-Symmetry

- *Objects* hide their data behind abstractions and expose functions that operate on that data;

- *Data structures* expose their data and have no meaningful functions;

Listing 6-5 (Procedural): `Geometry` is a(n operator) class which acts on three shape classes. The shape classes (`Square` , `Rectangle`, `Circle`) are simply data structures without any behavior.

Listing 6-6 (Polymorphic): There is no geometry operator and each of the shape classes implements its own `area` function.

Note that the procedural and polymorphic (OO) shapes are diametrically opposed! In the former, the shapes were strict data structures without any behavior. In the latter, the behavior (i.e, calculating an area) is built into the shape classes. In the former, including a new data structure require a change in the `Geometry` operator, but not on the existing data structures. In the latter, once we add a new data structure, we are responsible for its operation.

Fundamental dichotomies On one hand:

- Procedural code (e.g, using data structures) makes it easy to add new functions without changing the existing data structures. Note that one would not have to modify any of the other existing shapes, for example.
- OO code makes it easy to add new classes without changing existing functions.

On the other hand:

- Procedural code makes it hard to add new data structures because all functions must change. For example, modifying `area`, `perimeter`, `centroid` to include the `Triangle` data structure.
- OO code makes it hard to add new functions because all the classes must change. For example, modifying the `area` function across all the different Shape classes

Conclusion:

- When adding new data types, probably OO is more appropriate.
- When adding new functions, probably procedural code with data structures will be more appropriate.

Law of Demeter

A module should not know about the innards of the objects it manipulates. More precisely, a method f of a class C should only call methods of:

- C;
- An object created by f;
- An object passed as an argument to f;
- An object held in an instance variable of C;

The method should *not* invoke methods on objects that are returned by any of the allowed functions. *Talk to friends, not to strangers*

Train Wrecks

```
... = ctxt.getOptions().getScratchDir().getAbsolutePath()
```

This kind of code is often called a train wreck because it will look like a bunch of coupled train cars. These type of calls are *sloppy* and should be avoided. It is better to split it into three lines, the first defining a variable from `ctxt.getOptions()` (with the others following).

Whether it is a violation of the Demeter principle, depends on if `ctxt` and the following are objects or just data structures. If they are objects, then their internal structure should be hidden. If they are data structures, then the internal structure is naturally exposed and the law does not apply.

Part of the confusion also comes from the use of accessor functions rather than just returning parameters.

Hybrids

Avoid creating hybrids which are half object and half data structure because these are hard to maintain.

Hiding Structure

It is better to refactor the code such that the object `ctxt` does something rather than accessing several things of it in order to then do something. *I.e, the doing should be internal because we are accessing things which are internal in nature!*. In other words, `ctxt` should be responsible for doing all the steps, `ctxt.get_all()`.

Data Transfer Objects

This is a class with public variables and no functions. These are very useful structures, especially when communicating with databases or parsing messages from sockets and so on.

Somewhat more common is the "bean" form shown in Listing 6-7, which have private variables manipulated by getters and setters.

Active records

These are data structures with public (or bean-accessed) variables, but with navigational methods like `safe` and `find`. Typically are direct translations from database tables or other data sources.

Chapter 7: Error Handling

Error handling is important, but if it obscures logic, then it is wrong.

Write your `try-except-finally` statement first

Try blocks are like transactions. The except should leave the program in a consistent state, no matter what happens in the try. For this reason it is good practice to start with a try-except-finally statement when you are writing code that could throw exceptions.

Define Exception Classes in Terms of a Caller's Needs

There are many ways to classify errors. However, when we define exception classes in an application, our most important concern should be *how are they caught.*

One way this can be done neatly is by using *wrappers.* Wrapping third-party APIs is a best practice: In doing so, one minimizes the dependencies, because one can choose to move to a different library without much penalty. It is also much easier to mock code when you need to.

Define the normal flow

Guarantee that the class, for example, returns always the same object type.

Do not return Null

When returning `Null`, one is essentially creating downstream work and fostering problems upon our callers.

- When tempted to return `Null` from a method, consider throwing an exception or returning a Special Case object instead (see previous section).
- When calling a null-returning method from a third party API, consider wrapping the method with a method that either throws an exception or returns a Special Case object.

Do not pass Null

This is even worse than returning `Null`.

Conclusion

Error handling is an important but separate concern from main logic and these should be viewed independently.

Chapter 8: Boundaries

Using third-party code

There is a conflict of interests between the provider and the user of an interface:

- The providers strive for broad applicability;
- Users, however, want an interface that is focused on their needs;

The advice is not to pass any interface at a boundary around your system. If this is being used, keep it inside a class, or close family of classes, where it is used. Avoid returning it from, or accepting it as an argument to, public APIs.

Chapter 9: Unit tests

The three laws of Test Driven Development (TDD)

Write unit tests before you write the productive code. But this is just the beginning:

- *First law*: You may not write production code until you have written a failing unit test;
- *Second law*: You may not write more than one unit test to fail;
- *Third law*: You may not write more production code than is sufficient to pass the current test failing;

However, if you do end up following this methodology, you end up with a lot of tests.

Keeping tests clean

Test code is as important as production code!

Test enable the -ilities

Unit tests keep the code flexible, maintainable and reusable. *If you have tests, you do not fear making changes to the code.* Having an automated suit of unit tests that cover the production code is the key to keeping your design and architecture as clean as possible.

Clean tests

Emphasis on readability! This comes from clarity, simplicity and density of expression.

Use the *Build-Operate-Check* pattern:

1. Build the test data;
2. Operate on said data;
3. Checks that the operation yielded the expected results;

One assert per test (if you can)

Single concept per test

Minimize the number of asserts per concept and test just one concept per test function.

F.I.R.S.T

- *Fast*: Tests should run fast;
- *Independent*: Tests should not depend on each other;
- *Repeatable*: Tests should be repeatable in any environment;
- *Self-validating*: I.e, they should have a boolean output, thereby allowing for a quick comparison;
- *Timely*: Tests need to be written in a timely fashion, i.e, before the production code that makes them pass. Otherwise, you might come to the conclusion that the production code is too hard to test after the fact;

Chapter 10: Classes

Class organization

(Java convention) A class should begin with a list of variables in the following order:

1. Public static constants;
2. Private static variables;
3. Private instance variables;
4. (Avoid this) public variables;

Then, public functions. The private utility functions should be right after the public function that calls it.

Classes should be small

The name of the class should describe what responsibilities it fulfills. *If we cannot derive a concise name for a class, then it is likely too large.*

Single responsibility principle

A class or module should have one, and only one, reason to change - the thing they are responsible for.

Trying to identify responsibilities (i.e, reasons to change) often helps us recognize and create better abstractions in our code. Code should be built out of many small classes, with each representing a single responsibility (and therefore, one reason to change) and collaborates with a few others to achieve the desired system behavior.

Note: This is in tension with *Philosophy of Software Design*.

Cohesion

Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. A class in which each variable is used by each method is maximally cohesive. On one hand, it is not advisable to write maximally cohesive classes. *On the other hand, we would like cohesion to be high, so that the methods and variables of the class are co-dependent and hang together as a logical whole.*

The strategy of keeping functions small and keeping parameter lists short can sometimes lead to a proliferation of *instance variables* that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to get out of the larger class.

You should try to separate the variables and the methods into two or more classes such that the new classes are more cohesive.

Maintaining cohesion results in many small classes

Breaking a large function into many smaller functions often gives us the opportunity to split several smaller classes out as well.

The program has been split into three main responsibilities.

- The main program is contained in the `PrimePrinter` class (file) all by itself. Its responsibility is to handle the execution environment and it will change if the method of invocation changes.
- The `RowColumnPagePrinter` knows all about how to format a list of number into pages with a certain number of rows and columns. If the formatting of the output needs to be changed, this is the class that would be affected.
- The `PrimeGenerator` class knows how to generate a list of prime numbers. *It is not meant to be instantiated as an object.* This can be seen by the call in `PrimePrinter`, `primes = PrimeGenerator.generate(NUMBER_OF_PRIMES)`, rather than `object = PrimeGenerator` followed by `object.generate(NUMBER_OF_PRIMES)`. This class would change if the algorithm for computing prime number changes.

The change between the two programs was made by writing a test suit that verified the precise behavior of the first program. This was then followed up with the refactoring to 3 classes. After each change the program was executed to ensure that the behavior had not changed. *Think of incremental improvements when refactoring.*

Organizing for change

That there are two reasons to change a class shows that the `Sql` class (in Listing 10-9) violates the Single Responsibility Principle. Private method behavior that applies *only* to a small subset of a class can be a useful heuristic for spotting potential areas for improvement.

In the listing 10-10, each public interface method defined in the previous `Sql` class is refactored out to its own derivative of the `Sql` class.

In an ideal system, we incorporate new features by extending the system, not by making modifications to existing code.

Isolating from change

There are:

- abstract classes (also known as interfaces), which represent concepts only;
- concrete classes, which contain implementation details (code) and;

One way to isolate from changing details is to use interfaces and abstract classes. Dependencies upon concrete details create challenges for testing our system.

Dependency inversion principle

Our classes should depend upon abstractions, not on concrete details. In particular:

1. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

When thinking between the interaction between a high-level and a low-level module, the interaction should be thought of as an *abstract interaction between them.*

Chapter 11: Systems

How to stay clean at higher levels of abstraction, the system level.

Separate constructing a system from using it

Software systems should separate the startup process, when the application objects are constructed and the dependencies are wired together, from the runtime logic which takes over after startup.

Separation from Main

One way to separate construction from use is simply to move all aspects of construction to `main` or modules called by main, and to design the rest of the system assuming that all objects have been constructed and wired up appropriately.

The main function builds the objects necessary for the system, then passes them to the application, which simply uses them. Note that the application has no knowledge of main or of its construction process. It simply expects that everything has been built properly.

Factories (with a slight detour for context)

See Figure 11.2

From here:

The *Factory Method* as a creational design pattern (related to the creation of objects) with a common interface. It separates the process of creating an object from the code that depends on the interface of the object.

The first step when you see complex conditional code in an application is to identify the common goal of each of the execution (or logical) paths. Then, you can follow the process of:

1. Create a common interface which can be used to replace each of the paths;
2. Then, provide separate implementations for each logical path (in this particular example, one implementation for 'JSON' format and one for 'XML');
3. Provide a *separate component* which decides on which of the concrete implementations to use from 2.

Refactored code - without any other changes This is basically step 2 described above.

```
class SongSerializer:
    def serialize(self, song, format):
        if format == 'JSON':
            return self._serialize_to_json(song)
        elif format == 'XML':
            return self._serialize_to_xml(song)
        else:
            raise ValueError(format)

    def _serialize_to_json(self, song):
        payload = {
            'id': song.song_id,
            'title': song.title,
            'artist': song.artist
        }
        return json.dumps(payload)

    def _serialize_to_xml(self, song):
        song_element = et.Element('song', attrib={'id': song.song_id})
        title = et.SubElement(song_element, 'title')
        title.text = song.title
        artist = et.SubElement(song_element, 'artist')
        artist.text = song.artist
        return et.tostring(song_element, encoding='unicode')
```

Basic implementation of Factory Method

```
class SongSerializer:
    def _get_serializer(self, format):
        if format == 'JSON':
            return self._serialize_to_json
        elif format == 'XML':
            return self._serialize_to_xml
        else:
            raise ValueError(format)
```

Note that we are not calling anything concrete, it just returns the TBD function objects `self._serialize_to_json` and `self._serialize_to_xml`.

```
class SongSerializer:
    def serialize(self, song, format):
        serializer = self._get_serializer(format)
        return serializer(song)

    def _get_serializer(self, format):
        if format == 'JSON':
            return self._serialize_to_json
        elif format == 'XML':
            return self._serialize_to_xml
        else:
            raise ValueError(format)

    def _serialize_to_json(self, song):
        payload = {
            'id': song.song_id,
            'title': song.title,
            'artist': song.artist
        }
        return json.dumps(payload)

    def _serialize_to_xml(self, song):
        song_element = et.Element('song', attrib={'id': song.song_id})
        title = et.SubElement(song_element, 'title')
        title.text = song.title
        artist = et.SubElement(song_element, 'artist')
        artist.text = song.artist
        return et.tostring(song_element, encoding='unicode')
```

The `serialize` method is the application code that depends on an interface to complete its task.

This is referred to as the *client component* of the pattern. The interface defined is referred to as the product component. In our case, the product is a function that takes a `Song` and returns a string representation.

The `_serialize_to_json` and `_serialize_to_xml` methods are concrete implementations of the product. Finally, the `_get_serializer` method is the creator component. The creator decides which concrete implementation to use.

Second go at the code Because we started with some existing code, all the components of Factory Method are members of the same class `SongSerializer`. But because none of the added methods use `self`, you can factored these out of the class.

```
class SongSerializer:
    def serialize(self, song, format):
        serializer = _get_serializer(format)
        return serializer(song)

def _get_serializer(format):
    if format == 'JSON':
        return _serialize_to_json
    elif format == 'XML':
        return _serialize_to_xml
    else:
        raise ValueError(format)

def _serialize_to_json(song):
    payload = {
        'id': song.song_id,
        'title': song.title,
        'artist': song.artist
    }
    return json.dumps(payload)

def _serialize_to_xml(song):
    song_element = et.Element('song', attrib={'id': song.song_id})
    title = et.SubElement(song_element, 'title')
    title.text = song.title
    artist = et.SubElement(song_element, 'artist')
    artist.text = song.artist
    return et.tostring(song_element, encoding='unicode')
```

A client (`SongSerializer.serialize()`) depends on a concrete implementation of an interface. It requests the implementation from a creator component (`get_serializer()`) using some sort of identifier (`format`).

The Factory Method should be used in every situation where an application (client) depends on an interface (product) to perform a task and there are multiple concrete implementations of that interface. This requires a parameter (in the case above, format) which can identify the correct implementation and *use it in the creator (i.e, the get) to decide on the concrete implementation.*

```
# In serializers.py

import json
import xml.etree.ElementTree as et

class ObjectSerializer:
    def serialize(self, serializable, format):
        # serializable can be a song, a playlist or an album
        # format identifies the concrete implementation
        serializer = factory.get_serializer(format) # <--- CREATOR
        serializable.serialize(serializer)
        return serializer.to_str()

class JsonSerializer:
    def __init__(self):
        self._current_object = None

    def start_object(self, object_name, object_id):
        self._current_object = {
            'id': object_id
        }

    def add_property(self, name, value):
        self._current_object[name] = value

    def to_str(self):
        return json.dumps(self._current_object)


class XmlSerializer:
    def __init__(self):
        self._element = None

    def start_object(self, object_name, object_id):
```

```python
        self._element = et.Element(object_name, attrib={'id': object_id})

    def add_property(self, name, value):
        prop = et.SubElement(self._element, name)
        prop.text = value

    def to_str(self):
        return et.tostring(self._element, encoding='unicode')
```

The example defines the `Serializer` interface to be an object that implements the following methods or functions:

```python
def start_object(object_name, object_id):
    pass
def add_property(name, value)
    pass
def to_str():
    pass
```

```python
# In songs.py

class Song:
    def __init__(self, song_id, title, artist):
        self.song_id = song_id
        self.title = title
        self.artist = artist

    def serialize(self, serializer):
        serializer.start_object('song', self.song_id)
        serializer.add_property('title', self.title)
        serializer.add_property('artist', self.artist)
```

After implementing the product (`Serializer`) and the client (`ObjectSerializer`), we can complete the factory implementation by providing the creator (which above was simply a function, `_get_serializer`). Classes can provide additional interfaces to add functionality and can be derived to customize behavior. Unless one has a very basic and static creator, one likely wants to implement it as a class and not a function.

These type of classes are called object factories.

```python
# In serializers.py

class SerializerFactory:
    def get_serializer(self, format):
        if format == 'JSON':
            return JsonSerializer()
        elif format == 'XML':
            return XmlSerializer()
        else:
            raise ValueError(format)

factory = SerializerFactory()
```

Adding new formats is trivial, just include a new `Serializer` which satisfies those 3 properties. Note, however, that you would also have to change the object factory by including a new condition in the `if /elif/else` statement. Therefore, it is better to re-write the `SerializerFactory` as to avoid this, by adding a `register_format` method:

```python
# In serializers.py

class SerializerFactory:

    def __init__(self):
        self._creators = {}

    def register_format(self, format, creator):
        self._creators[format] = creator

    def get_serializer(self, format):
        creator = self._creators.get(format)
        if not creator:
            raise ValueError(format)
        return creator()


factory = SerializerFactory()
factory.register_format('JSON', JsonSerializer)
factory.register_format('XML', XmlSerializer)
```

Not all objects are created equal It is important that the creator - in this case, the `ObjectFactory` - returns fully initialized objects. If it does not, then the client will have to complete the initialization and therefore use conditional code, thereby defeating the purpose of using this design pattern.

Imagine that the application wants to integrate with a service provided by Spotify. This service requires an authorization process where a client key and secret are provided for authorization. There are several challenges: each service is initialized with a different set of parameters. Also, Spotify and Pandora require an authorization process before the service instance can be created.

Separate object creation to provide common interface The creation of each concrete music service has its own set of requirements. Therefore a common initialization interface for each service implementation is not possible or recommended. The best approach is to define a new type of object that:

1. provides a general interface and
2. is responsible for the creation of a concrete service.

This object will be called a `Builder` and has all the logic to create and initialize a service instance. Our goal is to implement a Builder object for each of the supported services (Spotify, which uses an access code, and Pandora, which uses a key and secret).

```python
# In program.py

config = {
    'spotify_client_key': 'THE_SPOTIFY_CLIENT_KEY',
    'spotify_client_secret': 'THE_SPOTIFY_CLIENT_SECRET',
    'pandora_client_key': 'THE_PANDORA_CLIENT_KEY',
    'pandora_client_secret': 'THE_PANDORA_CLIENT_SECRET',
    'local_music_location': '/usr/data/music'
}
```

```python
# In music.py

class SpotifyService:
    def __init__(self, access_code):
        self._access_code = access_code

    def test_connection(self):
        print(f'Accessing Spotify with {self._access_code}')


class SpotifyServiceBuilder:
    def __init__(self):
        self._instance = None

    def __call__(self, spotify_client_key, spotify_client_secret, **_ignored):
        if not self._instance:
            access_code = self.authorize(
                spotify_client_key, spotify_client_secret)
            self._instance = SpotifyService(access_code)
        return self._instance

    def authorize(self, key, secret):
        return 'SPOTIFY_ACCESS_CODE'
```

The `call` method of `SpotifyServiceBuilder` is used to create and initialize the concrete `SpotifyService`. It specifies the required parameters and ignores any additional parameters through `**_ignored`. Once the access code is retrieved, it creates and returns the `SpotifyService` instance.

Note that the reason for both storing and returning `SpotifyService` so that `SpotifyServiceBuilder` only creates a new one the first time the service is requested (see `__call__` method).

```python
# In music.py

class PandoraService:
    def __init__(self, consumer_key, consumer_secret):
        self._key = consumer_key
        self._secret = consumer_secret

    def test_connection(self):
        print(f'Accessing Pandora with {self._key} and {self._secret}')
```

```python
class PandoraServiceBuilder:
    def __init__(self):
        self._instance = None

    def __call__(self, pandora_client_key, pandora_client_secret, **_ignored):
        if not self._instance:
            consumer_key, consumer_secret = self.authorize(
                pandora_client_key, pandora_client_secret)
            self._instance = PandoraService(consumer_key, consumer_secret)
        return self._instance

    def authorize(self, key, secret):
        return 'PANDORA_CONSUMER_KEY', 'PANDORA_CONSUMER_SECRET'
```

```python
# In music.py

class LocalService:
    def __init__(self, location):
        self._location = location

    def test_connection(self):
        print(f'Accessing Local music at {self._location}')


def create_local_music_service(local_music_location, **_ignored):
    return LocalService(local_music_location)
```

The `LocalService` just requires a location where the collection is stored to initialize the `LocalService`. A new instance is created every time the service is requested because there is no slow authorization process. Since the requirements are simpler, there is no need for a Builder class. Instead, a function returning an initialized `LocalService` is used. This function matches the interface of the `__call__` methods implemented in the builder classes.

A generic interface to object factory This can leverage the generic Builder interface to create all kinds of objects. It provides methods to:

1. register a Builder based on a key value, `register_builders` and
2. to create the concrete object instances based on the key, `create`.

```python
# In object_factory.py

class ObjectFactory:
    def __init__(self):
        self._builders = {}

    def register_builder(self, key, builder):
        self._builders[key] = builder
    def create(self, key, **kwargs):
        builder = self._builders.get(key)
        if not builder:
            raise ValueError(key)
        return builder(**kwargs)
```

The difference between `SerializerFactory` and `ObjectFactory` is that the interface of the latter supports creating any type of object. The builder parameter can be any object that implements the callable interface. This means a Builder can be a function, a class, or an object that implements `__call__`. The `create` method requires that additional arguments are specified as keyword arguments, which allows the `Builder` objects to specify the necessary parameters and ignore the rest. For example, `create_local_music_service()` specifies a `local_music_location` parameter and ignores the rest.

```
# In music.py
import object_factory

# Omitting other implementation classes shown above

factory = object_factory.ObjectFactory()
factory.register_builder('SPOTIFY', SpotifyServiceBuilder())
factory.register_builder('PANDORA', PandoraServiceBuilder())
factory.register_builder('LOCAL', create_local_music_service)
```

```
# In program.py
import music

config = {
    'spotify_client_key': 'THE_SPOTIFY_CLIENT_KEY',
    'spotify_client_secret': 'THE_SPOTIFY_CLIENT_SECRET',
    'pandora_client_key': 'THE_PANDORA_CLIENT_KEY',
    'pandora_client_secret': 'THE_PANDORA_CLIENT_SECRET',
    'local_music_location': '/usr/data/music'
}

pandora = music.factory.create('PANDORA', **config)
pandora.test_connection()

spotify = music.factory.create('SPOTIFY', **config)
spotify.test_connection()

local = music.factory.create('LOCAL', **config)
local.test_connection()

pandora2 = music.services.get('PANDORA', **config)
print(f'id(pandora) == id(pandora2): {id(pandora) == id(pandora2)}')

spotify2 = music.services.get('SPOTIFY', **config)
print(f'id(spotify) == id(spotify2): {id(spotify) == id(spotify2)}')
```

Specializing object factory to improve code readability Using the object factory directly may lead to some confusion. To avoid this, one can specialize a general purpose implementation and provide an interface concrete to the application context:

```python
# In music.py

class MusicServiceProvider(object_factory.ObjectFactory):
    def get(self, service_id, **kwargs):
        return self.create(service_id, **kwargs)


services = MusicServiceProvider()
services.register_builder('SPOTIFY', SpotifyServiceBuilder())
services.register_builder('PANDORA', PandoraServiceBuilder())
services.register_builder('LOCAL', create_local_music_service)
```

Dependency Injection

This is the application of Inversion of Control to dependency management.

Inversion of Control moves secondary responsibilities from an object to other objects that are dedicated to the purpose, thereby supporting the Single Responsibility Principle. In the context of dependency management, an object should not take responsibility for instantiating dependencies itself - instead, it should pass the responsibility to another “authoritative” mechanism, thereby inverting the control.

Because setup is a global concern, this authoritative mechanism will usually be either the main routine or a special-purpose container.

A good example of Dependency Injection See here. *This particular example is Constructor Injection (or constructor arguments)*

The Inversion-of-Control (IoC) pattern, is about providing any kind of callback, which “implements” and/or controls reaction, instead of acting ourselves directly (in other words, inversion and/or redirecting control to the external handler/controller). The Dependency-Injection (DI) pattern is a more specific version of IoC pattern, and is all about removing dependencies from your code.

```
Every DI implementation can be considered IoC, but one should
not call it IoC, because implementing Dependency-Injection is
harder than callback (Don't lower your product's worth by using
the general term "IoC" instead)
```

For the case of DI, imagine an application has a text-editor component, and one wants to provide spell checking. The (java) code would look something like this:

```java
public class TextEditor {

    private SpellChecker checker;
    public TextEditor() {
        this.checker = new SpellChecker();
    }
}
```

This creates a dependency between `TextEditor` and `SpellChecker`. In an IoC scenario one would instead do:

```
public class TextEditor {

    private IocSpellChecker checker;

    public TextEditor(IocSpellChecker checker) {
        this.checker = checker;
    }
}
```

In the first code example one instantiates `SpellChecker`, which means the `TextEditor` class directly depends on the `SpellChecker` class.

In the second code example, one creates an abstraction by having the `SpellChecker` dependency class in `TextEditor` constructor signature (not initializing dependency in class). This allows us to call the dependency then pass it to the `TextEditor` class like so:

```
SpellChecker sc = new SpellChecker(); // dependency
TextEditor textEditor = new TextEditor(sc);
```

Now the client creating the `TextEditor` class has control over which `SpellChecker` implementation to use because one is injecting the dependency into the `TextEditor` signature.

Note that just like IoC being the base of many other patterns, above sample is only one of many Dependency-Injection kinds, for example:

- *Constructor Injection*: where an instance of `IocSpellChecker` would be passed to constructor, either automatically or similar to above manually.
- *Setter Injection*: where an instance of `IocSpellChecker` would be passed through setter-method or public property.
- *Service-lookup and/or Service-locator*: where `TextEditor` would ask a known provider for a globally-used-instance (service) of `IocSpellChecker` type (and that maybe without storing said instance, and instead, asking the provider again and again).

Ok, let us go back to the text True Dependency Injection goes one step forward. The class takes no direct steps to resolve its dependencies and is completely passive. Instead, it provides *setter methods* or *constructor arguments* (see previous section) that are used to *inject* the dependencies.

During the construction process, the DI container instantiates the required objects (usually on demand) and uses the constructor arguments or setter methods provided to wire together the dependencies. Which dependent objects are actually used is specified through a configuration file or programatically in a special-purpose construction module.

Scaling up

Software systems are unique compared to physical systems. Their architectures can grow incrementally, ***if*** *we maintain the proper separation of concerns*

In Aspect-Oriented Programming, modular constructs called aspects specify which points in the system should have their behavior modified in some consistent way to support a particular concern. This specification is done using a succinct declarative or programmatic mechanism.

Using persistence as an example, you would declare which objects and attributes (or patterns thereof) should be persisted and then delegate the persistence tasks to your persistence framework. The behavior modifications are made *noninvasively* to target code by the AOP framework.

Optimize decision making

Best to postpone decisions until the last possible moment. This allows us to make informed decisions with the best possible information. A premature decision is one made with suboptimal knowledge. There is much less customer feedback, mental reflection on the project and experience with our implementation choices if we decide too soon.

Chapter 12: Emergence

Getting clean via emergent design

Kent Beck's four rules of Simple Design (in order of importance):

- Runs all the tests;
- Contains no duplication;
- Expresses the intent of the programmer;
- Minimizes the number of classes and methods;

Simple design rule 1: Runs all the tests

Making our systems testable pushes us toward a design where our classes are small and single purpose (i.e, that obey the single responsibility principle). The more tests we write, the more we continue to push toward things that are simpler to test. Therefore, making sure our system is fully testable helps us create better designs.

Simple design rules 2: Refactoring

We can keep our code and classes clean by incrementally refactoring the code. This can be done easily because our test suit is robust enough to guarantee that we can do these changes without breaking anything.

This refactoring can be done with 3 rules:

No duplication Lines of code that are similar can often be massaged to look even more alike so that they can be more easily refactored. As we extract commonality at the micro level, we start to recognize violations of the Single Responsibility Principle. Consequently, we might move a new extracted method to another class. *Understand how to achieve reuse in the small is essential to achieving reuse in the large.*

The *Template Method* pattern is a common technique for removing higher-level duplication. In this particular example, rather than having one class `VacationPolicy` that implements two methods `accrueUSDivisionVacation()` and `accrueEUDivisionVacation()`, we can define `VacationPolicy` and the method `accrueVacation` - itself a composition of `calculateBaseVacationHours()`, `alterForLegalMinimums()` and `applyToPayroll()` - and derive two classes (one specific to the US, another to the EU), where we override the US and EU specific logic in `alterForLegalMinimums()`.

Expressive The majority of the cost of a software project is in long-term maintenance. In order to minimize the potential for defects as we introduce change, it is critical for us to be able to understand what a system does. For that, we need explicit code.

- Choose good names;
- Keep functions and classes small;
- Use standard nomenclature;
- Write proper unit tests. A primary goals of these is to act as documentation by example;

Chapter 13: Concurrency

Concurrency is a decoupling strategy. It helps decouple *what* gets done from *when* it gets done.

Myths and Misconceptions

- Concurrency does not always improve performance. It can improve performance, but *only* when there is a lot of waiting time that can be shared between multiple threads or processors.
- The design of a concurrent algorithm can be very different from the design of a single-threaded system. Decoupling *what* from *when* has usually a large effect on the structure of the system.

Challenges

The splitting of the computation into multiple threads can lead creates incorrect paths which lead to unexpected results.

Concurrency defense principles

SRP Concurrency design is complex enough to be a reason to change in its own right and therefore deserves to be separated from the rest of the code. Points to consider:

- Concurrency-related code has its own life cycle of development, change and tuning;
- It has its own challenges, different and often more difficult than non-concurrency code;
- The number of ways in which miswritten concurrency-based code can fail makes it challenging enough without the added burden of surrounding application code.

Recommendation: Keep the concurrency related code separated from the other code.

Corollary: Limit the scope of data Two threads modifying the same field of a shared object can interfere with each other, therefore causing unexpected behavior. One solution is to use the `synchronized` keyword to protect a critical section in the code that uses the shared object (the python equivalent would be a `threading.Lock()`). *Note that this should not be overdone.*

Recommendation: Take data encapsulation to heart, severely limit the access of any data that may be shared.

Corollary: Use copies of data Avoid sharing data and treat data sources as read only. In other cases, one can copy objects, collect the results from multiple threads in these copies and then merge the results in a single thread.

Corollary: Threads should be as independent as possible Write your threaded code such that each thread exists in its own independent world, sharing no data with any other thread. *Each thread processes one client request, with all of the required data coming from an unshared source and stored as local variables.*

Recommendation: Attempt to partition data into independent subsets that can be operated on by independent threads, possibly in different processors.

Know your execution models

There are several different ways to partition behavior in a concurrent application. To discuss these it is important to understand some basic definitions:

- *Bound resources*: resources of a fixed size or number used in a concurrent environment. Examples include database connections and fixed-size read/write buffers;
- *Mutual exclusion*: only one thread can access shared data or a shared resource at a time;
- *Starvation*: one thread (or group of) is prohibited from proceeding for an excessively long time or forever. For example, always letting fast-running threads through first could starve out longer running threads if there is no end to the fast-running threads;
- *Deadlock*: two or more threads waiting for each other to finish. Each thread has a resource that the other thread requires and neither can finish until it gets the other resource;
- *Livelock*: threads in lockstep, each trying to do work but finding another in the way. Due to resonance, threads continue trying to make progress but are unable to, for an excessively long time (or forever);

Producer-consumer One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it. The *queue* between the producers and the consumers is a *bound resource*. This means that producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume.

Readers-writers When you have a shared resource that primarily serves as the source of information for readers, but which is occasionally updated by writers, throughput is an issue. This can cause starvation and the accumulation of stale information. Allowing updates can impact throughput. Writers tend to block many readers for a long period of time, thus causing throughput issues.

The challenge is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput and avoiding starvation. A simple strategy makes writers wait until there are no readers before allowing an update.

Beware dependencies between synchronized methods

If there is more than one synchronized method on the same shared class, then your system may be written incorrectly. *Recommendation*: Avoid using more than one method on a shared object.

There is times where this must be the case. There are three ways to make the code correct:

- *Client*-based locking: have the client lock the server before calling the first method and make sure that the lock extends to the code calling the last method;
- *Server*-based locking: within the server, create a method that locks the server, calls all the methods, then unlocks. Have the client call the new method;
- *Adapted server*: create an intermediary that performs the locking.

Keep synchronized sections small

Locks are expensive because they create delays and add overhead. Our goal is therefore to design code with as few critical sections as possible.

Recommendation: Keep your synchronized sections as small as possible.

Writing correct shut-down code is hard

Graceful shutdown can be hard to get right, with common problems involving deadlock, with threads waiting for a signal to continue that never comes.

Recommendation: Think about shut-down early and get it working early. It is going to take longer than you expect. Review existing algorithms because this is probably harder than you think.

Testing threaded code

Recommendation: Write tests that have the potential to expose problems and then run them frequently, with different programmatic configurations and system configurations and load. If the tests ever fail, track down the failure. Do not ignore a failure just because the tests pass on a subsequent run.

In more detail:

- Treat spurious failures as candidate threading issues. *Do not ignore system failures as one-offs*;
- Get your non-threaded code working first;
- Make your threaded code pluggable *so that you can run it in various configurations*;
- Run with more threads than processors *to encourage task swapping*. The more frequently tasks swap, the more likely you are to encounter code that is missing a critical session or causes deadlock;
- Run on different platforms;
- Instrument your code to try and force failures. *To force a failure, affect the order of execution*. Instrumentation can then be done in two ways:
  - Hand-coded;
  - Automated, using an aspect-oriented framework;

Chapter 17: Smells and heuristics

These are things you *should not do* - code smells.

Comments

1. Inappropriate information: Comments should not hold information that is better held by a different kind of system (source code control, issue tracking, …);
2. Obsolete: It is best not to write a comment that will become obsolete;
3. Redundant: If it is describing something that adequately describes itself. *Comments should say things that code cannot say for itself*;
4. Poorly written comment;
5. Commented-out code;

Environment

1. Project build requires more than one step;
2. Running tests requires more than one step;

Functions

1. Too many arguments: Functions should have few arguments;
2. Output arguments: These are counter-intuitive. If your function changes the state of something, have it change the state of the object it is called on;
3. Boolean arguments: These mean that the function does more than one thing. Confusing and should be eliminated;
4. Dead function: Methods which are never called should be discarded;

General

1. Multiple languages in one source file;
2. Obvious behavior is unimplemented: Consider a function that translates the name of a day to a enum which represents the day. We would expect "Monday" to be translated to `DAY.MONDAY`. We would also expect the common abbreviations to be translated, and we would expect the function to ignore case. *When an obvious behavior is not implemented, readers and users can no longer depend on their intuition about function names;*
3. Incorrect behavior at the boundaries: Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm. *Do not rely on your intuition.* Look for every boundary condition and write a test for it;
4. Overridden safeties;
5. Duplication: Every time you see duplication in the code it represents a missed opportunity for abstraction. That duplication could probably come a subroutine or perhaps another class outright. *By folding the duplication into such an abstraction, you increase the vocabulary of the language of your design.*
    - When there is an `if/else` chain that appears again and again in various modules, these should be replaced by polymorphism;
    - Modules which have similar algorithms but do not share the same code should be replaced and addressed with `Template` or `Strategy` patterns;
6. Code at the wrong level of abstraction: All the lower level concepts should be in the derivatives and all the higher level concepts should be in the base class. This generalizes to source files, components and modules;
7. Base classes depending on their derivatives: In general, base classes should know nothing about their derivatives;
8. Too much information: A well-defined interface does not offer very many functions to depend upon, so coupling is low. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better. *Help keep coupling low by limiting information*;
9. Dead code: Code that it is not executed. Remove it from the system;
10. Vertical separation: Variables and functions should be defined close to where they are used. Local variables should be declared just above their first usage - and should have a small vertical scope. Private functions should be defined just *below* their first usage;
11. Inconsistency: Do all similar things in the same way;
12. Artificial coupling: Things that do not depend on each other should not be artificially coupled. In general, an artificial coupling is a coupling between two modules that serves no direct purpose. It is the result of putting something in a temporarily convenient - though inappropriate - location;
13. Feature envy: The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. *See example page 324.* There are, however, situations where feature envy is preferable to extending a class in order to avoid it. In the example of page 325, including it would violate the Single Responsibility Principle, the Open Closed Principle and the Common

Closure Principle;

14. Selector arguments: Selector arguments are just a lazy way to avoid splitting a large function into several smaller ones. *See example 326*;
15. Obscured intent: Do not try to make code as short as possible. You might be condensing code at the expense of readability;
16. Misplaced responsibility: Use the principle of least surprise to decide where to put code;
17. Use explanatory variables: One of the more powerful ways to make a program readable is to break up calculations up into intermediate values that are held in variables with meaningful names;
18. Function names should say what they do;
19. Understand the algorithm: Before you consider yourself done with a function, make sure you understand how it works. Often the best way to gain this knowledge is to refactor the function into something that it is so clean and expressive that it is obvious how it works;
20. Make logical dependencies (between modules) physical;
21. Prefer polymorphism to `if/else`;
22. Replace magic numbers (i.e, hard-coded constants) to named constants;
23. Be precise when you make a decision in your code. Know how you have made it and how will you deal with any exceptions. *Ambiguities and imprecision in code are either a result of disagreements or laziness. In either case, they should be eliminated;*
24. Structure over convention;
25. Encapsulate conditionals: Extract functions to explain the intent of the conditional;
26. Avoid negative conditionals;
27. *Functions should only do one thing*;
28. Hidden temporal couplings: Expose the temporal coupling by creating a bucket brigade (see page 334). Every function produces a result that the next function needs, so there is no reasonable way to call them out of order;
29. Do not be arbitrary;
30. Encapsulate boundary conditions;
31. Functions should descend only one level of abstraction: The statements within a function should all be written at the same level of abstraction, which should be one level below the operation described the name of the function;
32. Keep configurable data at high levels: If you have a constant - such as a default or a configuration that is known and expected at a high level of abstraction - *do not bury it in a low-level function.* Expose it as an argument to that low-level function called from the high-level function. *The lower levels of the application do not own the values of these constants;*
33. Avoid transitive navigation *(Law of Demeter)*: Avoid modules to know much about their collaborators. If A collaborates with B and B with C, we do not want modules A and C to know about each other;

Names

1. Choose descriptive names;
2. Choose names at the appropriate level of abstraction: do not pick names

that communicate implementation, choose names that reflect the level of abstraction of the class that you are working in. *Making code more readable requires a dedication to continuous improvement*;
3. Use standard nomenclature when possible;
4. Unambiguous names;
5. Use long names for long scopes: the length of a name should be related to the length of the scope.;
6. Names should describe side-effects: Names should describe everything that a function, variable or class is or does;

See Pages 440 and 441 for Cross References of Heuristics