Program Development Guidelines

Quality comes not from inspection, but from improvement of the production process
- W. Edwards Deming

Design

D1 Write code that communicate its purpose and intent (readability)
D2 Code Refactoring
D3 Favor object composition over class inheritance
D4 Programming to an interface, not an implementation (but DON'T overdo it)
D5 Be SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion)
D6 Premature optimization is the root of all evil

Coding

C1 Use intention-revealing names
C2 Use enum instead of constants to define a finite set of values
C3 Use exceptions rather than return codes
C4 Use unchecked exceptions
C5 Don't return null
C6 Prefer (and keep) debugging log messages to use of debuggers
C7 Crash Early

Testing

T1 Test everything that could possibly break (but NOT trivial accessors)
T2 Put the test class in the same package as the class under test, in a "test" folder
T3 Name the test methods with [methodName]_Should[ExpectedBehavior]_When[StateUnderTest]

D1 Write code that communicate its purpose and intent (readability)

Bugs sneak in when code need to be changed without a complete and clear understanding, it happens when features were added to a piece of software by maintenance programmers (or author programmers in their "maintenance mode") within a tight timeframe.

Programs must be written for people to read

To some people who understand the craft of programming so well, programming is a way to communicate one's thoughts through code: C*omputer language is not just a way of getting a computer to perform operations but rather that it is a* ***novel formal medium for expressing ideas about methodology.*** *Thus,* ***programs must be written for people to read****, and only incidentally for machines to execute.* From *The Structure and Interpretation of Computer Programs, Preface to the First Edition* [AS96]

Good programmers write code that humans can understand

It is fast and easy to work with readable code, that's why Martin Fowler use the ability to write readable code to measure the competence of a programmer: *Any fool can write code that a computer can understand,* ***good programmers write code that humans can understand****.* [FBBOR99] p.15.

D2 Code Refactoring

To quote the original sermon on structured programming by Edsger Dijkstra: *The extent to which the **program correctness** can be established is not purely a function or the program's external specifications and behavior but **depends critically on its internal structure**.* [Dij70] p.5

Refactor the code to make program easier to read and therefore easier to maintain

*Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet **improves its internal structure**. It is a disciplined way to clean up code that **minimizes the chances of introducing bugs.** In essence when you refactor you are **improving the design of the code after it has been written.***

*"Improving the design after it has been written." That's an odd turn of phrase. In our **current understanding of software development we believe that we design and then we code.** A good design comes first, and the coding comes second. Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The code **slowly sinks from engineering to hacking.***

*Refactoring is the opposite of this practice. With refactoring you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. You move a field from one class to another, pull some code out of a method to make into its own method, and push some code up or down a hierarchy. Yet the cumulative effect of these small changes can radically improve the design. It is the exact **reverse of the normal notion of software decay.***

With refactoring you find the balance of work changes. You find that design, rather than occurring all up front, occurs continuously during development. You learn from building the system how to improve the design. The resulting interaction leads to a program with a design that stays good as development continues. From the preface of *Refactoring: Improving the Design of Existing Code* [FBORR99]

Good designs are easy to test and difficult to mess up

Tom DeMarco et al put their insight about software design improvement in *Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior* [DHLMRR08]:

*…be capable and willing to look in detail at your people's designs, and be aware enough to see quality when it's there. Doing this for even the shortest time will quickly convince you that the gold-plating argument is a red herring; no design is made better in any way by piling on added features or glitz. Rather, **what enhances a design's aesthetic is what is taken away.** The best designs are typically spare and precisely functional, **easy to test and difficult to mess up when changes are required.** Moreover, they make you feel that there could be no better way to achieve the product's assigned functionality.*

D3 Favor object composition over class inheritance

This is an excerpt from *Design Patterns: Elements of Reusable Object-Oriented Software,* [GHJV96] *pp. 19-20*:

Inheritance breaks encapsulation

But class inheritance has some disadvantages too. First, you can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time. Second, and generally worse, parent classes often define at least part of their subclasses' physical representation. Because inheritance exposes a subclass to details of its parent's implementation, ***it's often said that "inheritance breaks encapsulation". The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will for the subclass to change.***

Implementation dependencies can cause problems when you're trying to reuse a subclass. Should any aspect of the inherited implementation not be appropriate for the new problem domains, the parent class must be rewritten or replaced by something more appropriate. This dependency ***limits flexibility and ultimately reusability****. One cure for this is to inherit only from abstract classes, since they usually provide little or no implementation.*

Object composition is defined dynamically at run-time through objects acquiring references to other objects. Composition requires objects to respect each other's interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others. But there is a payoff. Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type. Moreover, because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependencies.

Object composition has another effect on system design. ***Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task****. Your classes and* ***class hierarchies will remain small and will be less likely to grow into unmanageable monsters.*** *On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.*

That leads us to our second principle of object-oriented design:

Favor object composition over class inheritance

Ideally you shouldn't have to create new components to achieve reuse. You should be able to get all the functionality you need by assembling existing components through object composition. But this is rarely the case, because the set of available components is never quite rich enough in practice. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and object composition thus work together.

Beware of overusing inheritance

Nevertheless, our experience is that designers ***overuse inheritance as a reuse technique*** *and designs are often made more reusable (and simpler) by depending more on object composition.*

D4 Programming to an interface, not an implementation [GHJV96] p.18

Beware of overusing interface

This is perhaps a more well received concept, for example, the Spring Framework encourage the use of this pattern, it is often to see @Service usage examples with @Service applied on an interface reference, and use Spring Dependency Injection (DI) to inject an implementation instance. It is flexible when you need another implementation for the interface, just change the declaration in xml file to ask Spring to inject another implementation instance without recompiling the source code, at the cost of an additional interface class.

However, you would not do it extensively for every class used in your modules, you would not create a Date interface, so that we can inject another Date implementation just in case. (Java SE 8 indeed provides another set of date and time classes implementation, JSR-310)

More likely is when your code need some bug fix or enhancement, often require adding methods or parameters so that it could not be done by just injecting another implementation. Ask yourself, how many interfaces at your code base having only one implementation? How often you need to fix the very one implementation and that change propagates back to its interface? Unstable interfaces are only burdens for maintenance.

Modern mock framework don't need interface to mock

Another reason for having interfaces is for testability. Before the invention of modern mocking library, interfaces are required to implement mock objects, for example, jMock can only mock interfaces. However, the invention of modern mocking library like Mockito make concrete class mocking possible, thus interfaces should be exist solely for testability.

For more about when not to apply this principle, see OCP in the section "D5 Be SOLID".

D5 Be SOLID

SOLID is a mnemonic acronym introduced by Michael Feathers [SOLID] for the “first five principles” named by Robert C. Martin. The principles are Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Dependency-Inversion Principle (DIP), and Interface-Segregation Principle (ISP). Martin dedicates 5 chapters, one chapter for each principle in his Jolt Award-winning book *Agile Software Development: Principles, Patterns and Practices*. [Mar03]

Single Responsibility Principle (SRP)

SRP requires that a class should have only one responsibility (reason to change). For example, if a class contains both business rules and persistence control, it has two responsibilities, hence violating the SRP.

Open-Closed Principle (OCP)

OCP requires that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. In general it is an application of “programming to an interface, not an implementation”

However, as Martin pointed out in his book:

*...**conforming to the OCP is expensive**. It takes development time and effort to create the appropriate abstractions. Those abstractions also increase the complexity of the software design. There is a limit to the amount of abstraction that that developers can afford. **Clearly, we want to limit the application of the OCP to changes that are likely.** How do we know which changes are likely? We do the **appropriate research**, we ask the **appropriate questions**, and we use our experience and common sense. And after all that, we **wait until the changes happen!*** [Mar03] p.105

*…apply abstraction only to those parts of the program that exhibit frequent change. **Resisting premature abstraction is as important as abstraction itself.*** [Mar03] pp.108-109

Liskov Substitution Principle (LSP)

LSP requires that subtypes must be substitutable for their base types.

Violating the LSP often results in the use of Run-Time Type Information (RTTI) in a manner that grossly violates the OCP. Frequently, an explicit if statement of if/else chain is used to determine the type of an object so that the behavior appropriate to that type can be selected, [Mar03] p.112

The following listing shows how LSP is violated:

```
void drawShape(Shape shape) {
      if (shape instanceof Square)
            ((Square) shape).draw();
      else if (shape instanceof Circle)
            ((Circle) shape).draw();
```

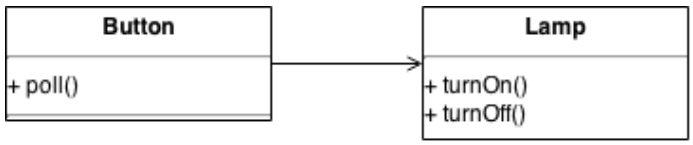
```
    }
```

The drawShape() violates OCP because it must be changed whenever new derivatives of Shape are created.

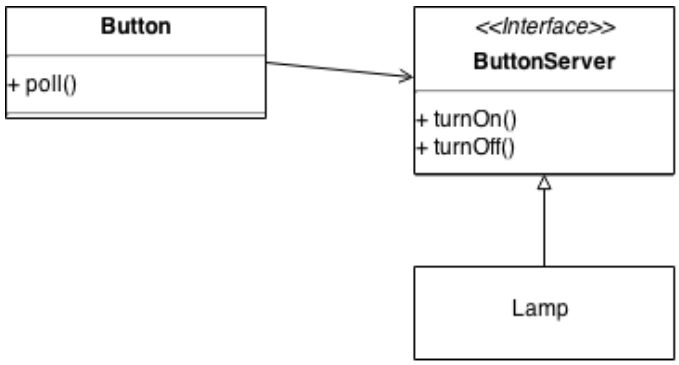
Dependency-Inversion Principle (DIP)

DIP requires that
a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
b. Abstractions should not depend on details. Details should depend on abstractions.

In short it is also an application of “programming to an interface, not an implementation”; consider the following solution that violates the DIP:



The following is a solution that conform to the DIP, a high-level module (Button) depends on an abstraction (ButtonServer), and detail (Lamp) depends on an abstraction:

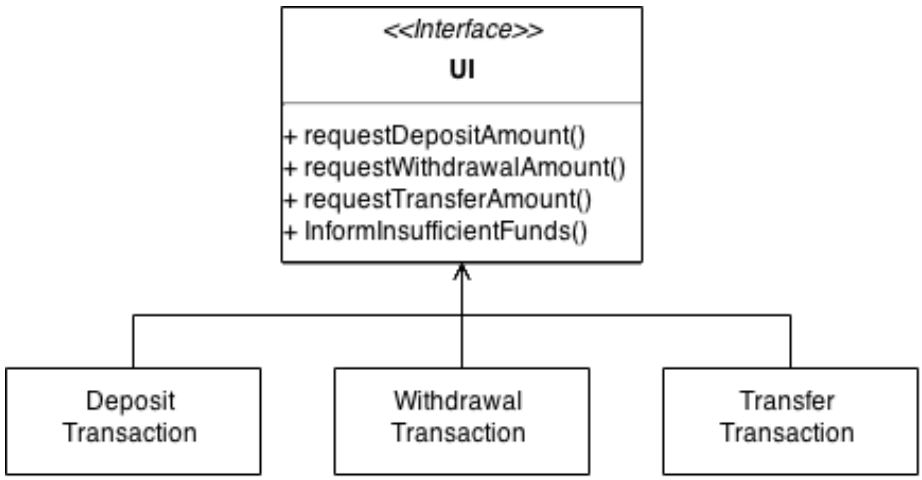


Similarly to OCP, “resisting premature abstraction” heuristic applies.

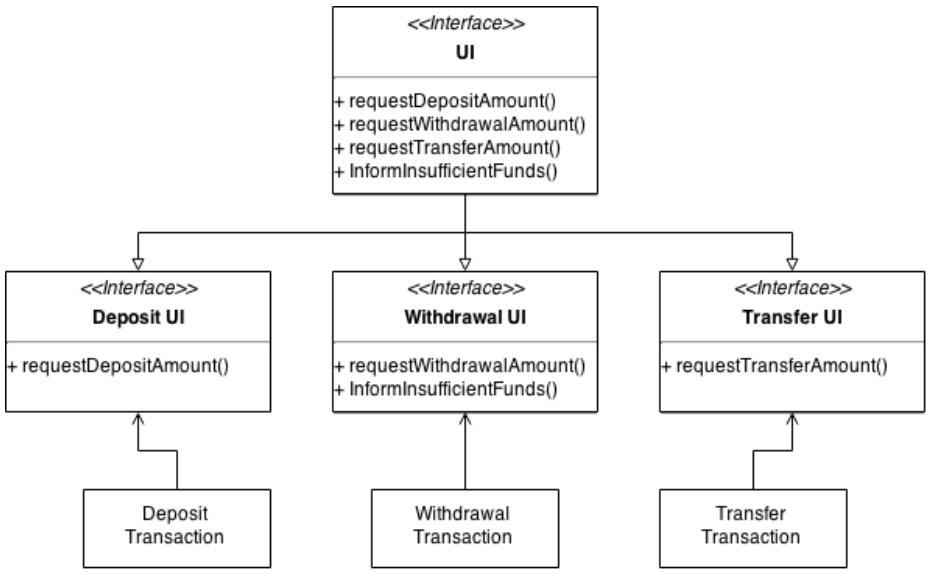
Interface-Segregation Principle (ISP)

ISP deals with the **disadvantages of “fat” interfaces**, it requires that **clients should not be forced to depend on methods that they do not use**. In other words, the “fat” interfaces can be broken up into groups of methods, each group serves a different set of clients.

The following diagram shows a “fat” interface:



With ISP applied:



However, Martin also points out in his book, ***As with all principles, care must be taken not to overdo it***. *The spectre of a class with hundreds of different interfaces, some segregated by client and others segregated by version, would be frightening indeed.* [Mar03] p.145

Martin was kind enough to put the articles (not exactly the same but very similar to those in his book) in the Object Mentor web site, check them out:
http://www.objectmentor.com/resources/articles/srp.pdf
http://www.objectmentor.com/resources/articles/ocp.pdf
http://www.objectmentor.com/resources/articles/lsp.pdf
http://www.objectmentor.com/resources/articles/isp.pdf
http://www.objectmentor.com/resources/articles/dip.pdf

D6 Premature optimization is the root of all evil

Needless complexity reduce maintainability

To quote from *Computer programming as an art* by Donald E. Knuth: *Another important aspect of program quality is the efficiency with which the computer's resources are actually being used. I am sorry to say that many people nowadays are condemning program efficiency, telling us that it is in bad taste. The reason for this is that we are now experiencing a reaction from the time when efficiency was the only reputable criterion of goodness, and programmers in the past have tended to be so preoccupied with efficiency that they have produced* ***needlessly complicated code****; the result of this unnecessary complexity has been that* ***net efficiency has gone down, due to difficulties of debugging and maintenance****. The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times;* ***premature optimization is the root of all evil*** *(or at least most of it) in programming.* [Knu74] p.671.

Dijkstra put it in this way: *My* ***refusal to regard efficiency considerations as the programmer's prime concern*** *is not meant to imply that I disregard them. On the contrary, efficiency considerations are recognized as one of the main incentives to modifying a logically correct program. My point, however, is that* ***we can only afford to optimize*** *(whatever that may be)* ***provided that the program remains sufficiently manageable****.* [Dij70] p.7.

Not an excuse for making bad choices

Don't use it as an excuse *to defend all sorts of choices, ranging from poor architectures, to gratuitous memory allocations, to inappropriate choices of data structures and algorithms, to complete disregard for variable latency in latency-sensitive situations*. [Duf10] However, think twice when you are going to spend a few hours or so to do an optimization.

C1 Use intention-revealing names

The name should tell you why it exists, what it does, and how it is used

From *Clean Code: A Handbook of Agile Software Craftsmanship*, [Mar09] p.18.

Choosing good names takes time but saves more than it takes. So take care with your names and change them when you find better ones. Everyone who reads your code (including you) will be happier if you do.

The name of a variable, function, or class, should answer all the big questions. It ***should tell you why it exists, what it does, and how it is used****. If a name requires a comment, then the name does not reveal its intent.*

```
int d; // elapsed time in days
```

The name d reveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

C2 Use enum instead of constants to define a finite set of values

Since Java 1.4 doesn't have enums, constants are used to define a finite set of values:

```
public static final int MONDAY = 1;
public static final int TUESDAY = 2;
public static final int WEDNESDAY = 3;
public static final int THURSDAY = 4;
public static final int FRIDAY = 5;
public static final int SATURDAY = 6;
public static final int SUNDAY = 7;
```

Do use enums in Java 5

Now that enums have been added to the language (since Java 5), don't keep using the old trick of constants. The meaning of ints can get lost. The meaning of enums cannot, because they belong to an enumeration that is named.

If integer values need to map to the enums, declare the enums with constructor for mapping the values:

```
public static enum DayOfWeek {
      MONDAY(1),
      TUESDAY(2),
      WEDNESDAY(3),
      THURSDAY(4),
      FRIDAY(5),
      SATURDAY(6),
      SUNDAY(7);

      DayOfWeek(int value) {
            this.value = value;
      }

      public static DayOfWeek fromValue(int value) {
            for (DayOfWeek d : DayOfWeek.values()) {
                  if (d.value == value)
                        return d;
            }
            throw new IllegalArgumentException("Invalid day of week value=" +
      value);
      }

      public final int value;
}
```

C3 Use exceptions rather than return codes

The following is an excerpt from *Clean Code: A Handbook of Agile Software Craftsmanship*, [Mar09] pp.104-105.

Back in the distant past there were many languages that didn't have exceptions. In those languages the techniques for handling and reporting errors were limited. You either set an error flag or returned an error code that the caller could check. The code in Listing 7-1 illustrates these approaches.

Listing 7-1:

```
public class DeviceController {
      …
      public void sendShutDown() {
            DeviceHandle handle = getHandle(DEV1);
            // Check the state of the device
            if (handle != DeviceHandle.INVALID) {
                  // Save the device status to the record field
                  retrieveDeviceRecord(handle);
                  // If not suspended, shut down
                  if (record.getStatus() != DEVICE_SUSPENDED) {
                        pauseDevice(handle);
                        clearDeviceWorkQueue(handle);
                        closeDevice(handle);
                  } else {
                        logger.log("Device suspended. Unable to shut down");
                  }
            } else {
                  logger.log("Invalid handle for: " + DEV1.toString());
            }
      }
      …
}
```

The problem with these approaches is that they clutter the caller. The caller must check for errors immediately after the call. Unfortunately, it's easy to forget. For this reason it is better to throw an exception when you encounter an error. The calling code is cleaner. Its logic is not obscured by error handling. Listing 7-2 shows the code after we've chosen to throw exceptions in methods that can detect errors.

Listing 7-2:

```
public class DeviceController {
      …
      public void sendShutDown() {
            try {
                  tryToShutDown();
            } catch (DeviceShutDownError e) {
                  logger.log(e);
            }
      }
```

```
        private void tryToShutDown() throws DeviceShutDownError {
              DeviceHandle handle = getHandle(DEV1);
              DeviceRecord record = retrieveDeviceRecord(handle);
              pauseDevice(handle);
              clearDeviceWorkQueue(handle);
              closeDevice(handle);
        }

        private DeviceHandle getHandle(DeviceID id) {
              …
              throw new DeviceShutDownError("Invalid handle for: " +
        id.toString());
              …
        }
        …
  }
```

Notice how much cleaner it is. This isn't just a matter of aesthetics. The code is better because two concerns that were tangled, the algorithm for device shutdown and error handling, are now separated. You can look at each of those concerns and understand them independently.

C4 Use unchecked exceptions

Problems with checked exceptions are **versioning and scalability issues**, and **breaking encapsulation**.

Checked exceptions lead to versioning and scalability issues

The following is an excerpt from *The Trouble with Checked Exception* [Hej03], a conversation with Anders Hejlsberg, who led the team that designed the C# language.

Let's start with versioning, because the issues are pretty easy to see there. Let's say I create a method foo that declares it throws exceptions A, B, and C. In version two of foo, I want to add a bunch of features, and now foo might throw exception D. It is a breaking change for me to add D to the throws clause of that method, because existing caller of that method will almost certainly not handle that exception. ***Adding a new exception to a throws clause in a new version breaks client code****. It's like adding a method to an interface. After you publish an interface, it is for all practical purposes immutable, because any implementation of it might have the methods that you want to add in the next version. So you've got to create a new interface instead. Similarly with exceptions, you would either have to create a whole new method called foo2 that throws more exceptions, or you would have to catch exception D in the new foo, and transform the D into an A, B, or C.*

The scalability issue is somewhat related to the versionability issue. In the small, checked exceptions are very enticing. With a little example, you can show that you've actually checked that you caught the FileNotFoundException, and isn't that great? Well, that's fine when you're just calling one API. The trouble begins when you start building big systems where you're talking to four or five different subsystems. Each subsystem throws four to ten exceptions. Now, each time you walk up the ladder of aggregation, you have this exponential hierarchy below you of exceptions you have to deal with. You end up having to declare 40 exceptions that you might throw. ***And once you aggregate that with another subsystem you've got 80 exceptions in your throws clause. It just balloons out of control****.*

In the large, ***checked exceptions become such an irritation that people completely circumvent the feature.*** *They either say, "throws Exception," everywhere; or—and I can't tell you how many times I've seen this—they say, "try, da da da da da, catch curly curly." They think, "Oh I'll come back and deal with these empty catch clauses later," and then of course they never do. In those situations,* ***checked exceptions have actually degraded the quality of the system in the large.***

Checked exceptions break encapsulation

The following is an excerpt from *Clean Code: A Handbook of Agile Software Craftsmanship*, [Mar09] pp.106-107.

The debate is over. For years Java programmers have debated over the benefits and liabilities of checked exceptions. When checked exceptions were introduced in the first version of Java, they seemed like a great idea. The signature of every method would list all of the exceptions that it could pass to its caller. Moreover, these exceptions were part of the type of the method. Your code literally wouldn't compile if the signature didn't match what your code could do. At the time, we thought that checked

exceptions were a great idea; and yes, they can yield some benefit. However, it is clear now that they aren't necessary for the production of robust software. ***C# doesn't have checked exceptions, and despite valiant attempts, C++ doesn't either. Neither do Python or Ruby. Yet it is possible to write robust software in all of these languages. Because that is the case, we have to decide—really—whether checked exceptions are worth their price.***

What price? ***The price of checked exceptions is an Open-Closed Principle violation.*** *If you throw a checked exception from a method in your code and the catch is three levels above, you must declare that exception in the signature of each method between you and the catch. This means that* ***a change at a low level of the software can force signature changes on many higher levels.*** *The changed modules must be rebuilt and redeployed, even though nothing they care about changed. Consider the calling hierarchy of a large system. Functions at the top call functions below them, which call more functions below them, ad infinitum. Now let's say one of the lowest level functions is modified in such a way that it must throw an exception. If that exception is checked, then the function signature must add a throws clause. But this means that every function that calls our modified function must also be modified either to catch the new exception or to append the appropriate throws clause to its signature. Ad infinitum. The net result is a cascade of changes that work their way from the lowest levels of the software to the highest!* ***Encapsulation is broken because all functions in the path of a throw must know about details of that low-level exception.*** *Given that the purpose of exceptions is to allow you to handle errors at a distance, it is a shame that checked exceptions break encapsulation in this way. Checked exceptions can sometimes be useful if you are writing a critical library: You must catch them. But in general application development the dependency costs outweigh the benefits.*

C5 Don't return null

A billion-dollar mistake

If methods don't return null, you don't need to check for null, nor forget to check for null, hence no NullPointerException. Tony Hoare apologized for inventing the null reference in 1965 and calls it **"My billion-dollar mistake"**. [Hoa09]

The problems of returning null

The following is an excerpt from *Clean Code: A Handbook of Agile Software Craftsmanship*, [Mar09] pp.110-111.

I think that any discussion about error handling should include mention of the things we do that invite errors. The first on the list is returning null. I can't begin to count the number of applications I've seen in which nearly every other line was a check for null. Here is some example code:

```
public void registerItem(Item item) {
      if (item != null) {
            ItemRegistry registry = peristentStore.getItemRegistry();
            if (registry != null) {
                  Item existing = registry.getItem(item.getID());
                  if (existing.getBillingPeriod().hasRetailOwner()) {
                        existing.register(item);
                  }
            }
      }
}
```

*If you work in a code base with code like this, it might not look all that bad to you, but **it is bad!** When we return null, we are essentially creating work for ourselves and foisting problems upon our callers. **All it takes is one missing null check to send an application spinning out of control.***

Did you notice the fact that there wasn't a null check in the second line of that nested if statement? What would have happened at runtime if persistentStore were null? We would have had a NullPointerException at runtime, and either someone is catching NullPointerException at the top level or they are not. Either way it's bad. What exactly should you do in response to a NullPointerException thrown from the depths of your application?

*It's easy to say that the problem with the code above is that it is missing a null check, but in actuality, the problem is that it has too many. **If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead.** If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.*

In many cases, special case objects are an easy remedy. Imagine that you have code like this:

```
List<Employee> employees = getEmployees();
```

```
if (employees != null) {
      for(Employee e : employees) {
            totalPay += e.getPay();
      }
}
```

Right now, getEmployees can return null, but does it have to? If we change getEmployee so that it returns an empty list, we can clean up the code:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
      totalPay += e.getPay();
}
```

Fortunately, Java has Collections.emptyList(), and it returns a predefined immutable list that we can use for this purpose:

```
public List<Employee> getEmployees() {
      if( .. there are no employees .. )
            return Collections.emptyList();
}
```

If you code this way, you will minimize the chance of NullPointerExceptions and your code will be cleaner.

Consider using **Null Object** pattern [Mar03] p.189, instead of returning null. Which is one of the aforementioned SPECIAL CASE object.

C6 Prefer (and keep) debugging log messages to use of debuggers

Ideally, the program's log messages should provide sufficient details for a programmer to deduce whether a program is working correctly when something wrong reported. If bugs were suspected, the log messages should provide sufficient details for a programmer to reproduce the bugs. Don't get me wrong, it's not saying we should put debug log all over the places, nor put them into `System.out.println()` so that the message only send to console and vanish after the program terminated.

As a personal choice, we tend ***not to use debuggers beyond getting a stack trace or the value of a variable or two.*** *One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding* ***output statements and self-checking code at critical places.*** *Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important,* ***debugging statements stay with the program; debugger sessions are transient.*** [KP99], p.119.

C7 Crash Early

You might have seen this trick to avoid NullPointerException (NPE) already, using `“s”.equals(s)` instead `s.equals(“s”)`, when the variable `s` is null, the former one returns false, the later one throws a NPE. The problems are, does a null value violate the preconditions? And is false a sensible default value for the situation? You could convince yourself that the variable s can’t be null, and choose to ignore it. However, as states by Murphy’s Law: “**Anything that can go wrong, will go wrong.**”

I have once witness some restricted resources have been opened for booking for more than two months, due to a misconfiguration, and the program logic ignore the null value so that a nonsensible default returns that permit the booking.

*One of the benefits of detecting problems as soon as you can is that you can crash earlier. And **many times, crashing your program is the best thing you can do. The alternative may be to continue, writing corrupted data to some vital database** or commanding the washing machine into its twentieth consecutive spin cycle. The Java language and libraries have embraced this philosophy. When something unexpected happens within the runtime system, it throws a RuntimeException. If not caught, this will percolate up to the top level of the program and cause it to halt, displaying a stack trace.* [HT99]

If the mentioned booking program logic throws an exception when it encounters an unexpected null, yes, someone might get blamed, but the blamed could be much easier to handle.

T1 Test everything that could possibly break

Let's begin with a little bit history about the automated unit testing. The most popular automated unit testing framework for Java, JUnit, was written by Kent Beck, who is also the creator of the Extreme Programming (XP) and Test Driven Development (TDD) software development methodologies [KentBeck]. XP suggested to *Test everything that could possibly break* [JAH00]. However, this statement might have been taken too far, as setters and getters could possibly break, so that they should be tested.

The following is an excerpt from *Extreme Programming Installed*, [JAH00] p.267.

The rule is to ***test everything that could possibly break. To save time, don't test things that couldn't possibly break.*** *There are more things that couldn't possibly break than you might imagine.*

Start conservatively on identifying things not to test. Until you're sure, test. But unit testing is "white box" testing. You look at the code when you write the test, and if the code can't break – don't test it.

Accessors can't break. There's no need to test them. *Unless, of course, you have a tendency to forget to write them, and no other test is going to find that they're missing. But wait – if no other test is going to access them, they shouldn't be there anyway. So probably you don't need to test accessors.*

T2 Put the test class in the same package as the class under test, in a “test” folder

You can place your tests in the same package and directory as the classes under test. These approaches allow the tests to access to all the public and package visible methods of the classes under test. Some developers have argued in favor of putting the tests in a sub-package of the classes under test (e.g. com.xyz.test). The author of this FAQ sees no clear advantage to adopting this approach and believes that said developers also put their curly braces on the wrong line. :-) [JUnit FAQ]

Usually, developers unit test the code using the main method or by executing the application. Neither of them is the correct approach. ***Mixing up production code with tests is not a good practice. It creates a code maintainability problem.*** *The best approach is to create a separate source folder for unit tests and put the test class in the same package as the main class. Usually, if a class name is TaxCalculator, its test should have the name TaxCalculatorTest.* [Ach13] p.8.

T3 Name the test methods with [methodName]_Should[ExpectedBehavior]_When[StateUnderTest]

Optionally insert underscores "_" before "Should" and "When" for readability, e.g. `[methodName]_Should[ExpectedBehavior]_When[StateUnderTest]`.

In JUnit 3, due to the lack of @Test, test methods have to prefix with "test". However in JUnit 4 with @Test you can name the test methods with anything.

Many test methods naming conventions were invented [Osh05] [Kum14], the common goal is to communicate the intent. Here are some examples: enquiryShouldLogRequestWhenInvoked, enquiryShouldSplitMultipleVenueCodesDelimitedByCommaWhenResponseCodeSuccessReturned, enquiryShouldRethrowExceptionWhenExceptionFromCmbcsCoreCaught.

The article by Roy Osherove also discussed how to name variables in test method to make it more readable. [Osh05]

References:

[Ach13] Test-Driven Development with Mockito, Sujoy Acharya, Packt Publishing, 2013.

[AS96] Structure and Interpretation of Computer Programs (Second Edition), Harold Abelson and Gerald Jay Sussman with Julie Sussman, The MIT Press, 1996.
http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-7.html#%_chap_Temp_4

[DHLMRR08] Robertson, Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior, Tom DeMarco, Peter Hruschka, Tim Lister, Steve McMenamin, James Robertson, Suzanne Dorset House, 2008.

[Dij70] Notes on Structured Programming (Second Edition), Edsger W. Dijkstra, 1972.
https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

[Duf10] The 'premature optimization is evil' myth, Joe Duffy, 2010.
http://joeduffyblog.com/2010/09/06/the-premature-optimization-is-evil-myth/

[FBBOR99] Refactoring: Improving the Design of Existing Code, Martin Fowler et al., Addison-Wesley, 1999

[GHJV96] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1996.

[Green] How To Write Unmaintainable Code
https://thc.org/root/phun/unmaintain.html

[Hej03] The Trouble with Checked Exception, Anders Hejlsberg, 2003
http://www.artima.com/intv/handcuffs.html

[Hoa09] Null References: The Billion Dollar Mistake, Tony Hoare, 2009
http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[HT99] The Pragmatic Programmer: From Journeyman to Master, Andrew Hunt, David Thomas, Addison-Wesley, 1999.

[JAH00] Extreme Programming Installed, Ron Jeffries, Ana Anderson, Chet Hendrickson, Addison-Wesley, 2000.

[JUnitFAQ] http://junit.org/faq.html

[KentBeck] Wikipedia, 2014-12-19.
http://en.wikipedia.org/wiki/Kent_Beck

[Knu74] Donald E. Knuth, "Computer programming as an art", ACM, 1974
http://delivery.acm.org/10.1145/370000/361612/a1974-knuth.pdf?ip=202.126.217.122&id=361612&acc=OPEN&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&CFID=458322274&CFTOKEN=48708513&__acm__=1418973821_73e6c3bd4e66c4fe0f1c890c4e42648a

[Kum14] 7 Popular Unit Test Naming Conventions, Ajitesh Kumar, 2014.
http://java.dzone.com/articles/7-popular-unit-test-naming

[KP99] The Practice of Programming, Brian W. Kernighan, Rob Pike, Addison-Wesley, 1999.

[Mar03] Agile Software Development: Principles, Patterns and Practices, Robert C. Martin, Pearson Education, 2003.

[Mar09] Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, Pearson Education, 2009.

[Osh05] Naming standards for unit tests, Roy Osherove, 2005.
http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html

[SOLID] Wikipedia, 2014-12-19
http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29