Lecture Notes,

CS345, Winter 2018

Class Design Guidelines

Review

Objects – capsules containing data and behavior

Classes – templates or “cookie cutters” for objects, specify both interface and implementation

Interfaces – specify object behavior without implementation

Behavior – what an object does, the external aspect of an object

Interface and Implementation – the external (“what”) and internal (“how”) of classes and objects

Inheritance – generalization and specialization, IS-A relationship

Composition – HAS-A relationship, using one object to provide behavior for another object

Big Question

We have this machine called Object-Oriented Programming. How do we use it? How do we do it right? Here are some more detailed questions:

- In a program, how much and/or what part of the program is included in a single class?
- When do you use interfaces versus classes?
- When is a class or interface too big or too small?
- How do you go about designing and constructing an object-oriented program?
- What are specialized classes of objects?
- How do construct and destroy objects?
- How do you do error handling?

This lecture contains heuristics for object-oriented design.

Heuristic: Model the Real World

Model real world systems. If you will, make the objects in your program correspond to the “real world” objects your program describes. Here are some ideas for how to do this:

- Do it the way you think about things
  - Game, like SimCity, has zones, roads, power plants, ...
  - Adventure game: Classes for words, actions, rooms, etc.
  - Email system: emails, mailboxes, etc.
  - Order processing system: customers, orders, money, etc.
- Design classes by creating models of these real world objects

        - What behavior does the real world object have?
            - Is that behavior important to your program?
                - Yes—your class needs to have that behavior.
                - No—ignore that behavior
                    - The real world is big and messy. You don't want to deal with any behavior you don't have to. If you do, that's more work for no benefit.
- Don't think of objects as events or functions that operate on the data in your program.
    - That's probably what you did in introductory programming.
- Think of objects as things that are responsible for implementing or providing a certain behavior.
    - The object contains the data and functions that are required to provide that behavior.
- Example:
    - Wrong: In a “SimCity like Game”, maintaining a table of how many people live and work in each zone. When you have to update population due to advancing time, you go through the table and apply some formula to each zone to update the values in that zone.
        - If you add a zone, you update the table to reflect that the new population information for that zone.
    - Right: Each zone maintains it's own information about what it's population, and how fast and under what condition it changes. Each zone can separately account for factors like pollution, police stations, etc. The game maintains a list of all zones. When time advances, each zone is called to update it's information. The total population of the city is computed by asking each zone to return its current population.
        - This approach allows for complex relationships to be modeled simply. For example, if population growth rates depend on the exact type of residences, low or high density, this can be easily accounted for.
- In general, interactions between objects will mimic interactions in the real world.

Responsibilities

Each object is responsible for something. In general, objects are responsible for either *doing* something or *knowing* something. For example,

- An object that is responsible for playing a piece of music (a doing responsibility) might
    - Be constructed with the source of the music
    - Have methods for pausing the music, changing the volume, etc.
    - Maintain the associated data, such as the current location in the stream, the volume, etc.
- An object that is responsible for maintaining a music catalog (a knowing responsibility) might
    - Be constructed as an empty object or by reading the catalog from a database
    - Have methods for adding items to and deleting items from the catalog
    - Have methods for searching the catalog
    - Maintain (a copy of) the catalog within the object

A music player application would use instances of both classes to find and play music.

Heuristic: Small Cohesive set of Responsibilities

Here's another version of the same heuristic: Single Responsibility Principle[1].

Ideally, each class should have a single responsibility. In some cases this is not feasible. However, a class should have a small set of cohesive responsibilities. (Cohesive means naturally or logically connected.)

Here's an example: A class that is responsible for generating and printing reports. That class has two distinct responsibilities. Better would be two separate classes, one that prepares the content of the report and one is responsible for formatting the content.

Another example: A class that is responsible for playing a piece of music, should not also be responsible for decoding music in different compression formats. That's a distinct responsibility that should be given to a separate set of objects (MP4 decoder, FLAC decoder, etc.). You need two (actually more than two) classes, one that is responsible for generating a stream, and one that is responsible for playing the generated stream.

In general if a class has too many or weakly related responsibilities, it should be divided into multiple classes.

Robert Martin describes this principle as “a class should only have a single reason to change” or “a class should only have a single source of change.” The idea is that every responsibility represents a reason that we might have to change the code. Ideally, when I have to change a class, I should be changing it in response to a single source of change. That, hopefully, prevents two different people changing the same class for two different reasons and stepping on each other in the process.

Interface and Implementation

By now you should know that the most important aspect of class design is identifying the external behavior that will be offered to clients of a class.

Two points:

1. A class should provide the behaviors that are required by the clients.
2. A class should only provide the behaviors required by the clients.

If a class has no external behaviors, it doesn't have a reason to exist.

Heuristic: Offer the Minimum Public Interface

Interfaces, meaning both an interface and the interface to a class shouldn't be too small and it shouldn't be too big. (Another “Goldilocks” principle.)

A class should offer the minimum public interface that is needed so that the clients can use it.

But, what does “needed” mean? Do we include convenience functions for actions that could be accomplished by the client. Answer: Maybe.

Implementing a class is a “business proposition”. There are costs associated with designing the interface, building and maintaining the class:

- The bigger the interface, the more expensive it is to build and maintain the class.

[1] http://en.wikipedia.org/wiki/Single_responsibility_principle

- A poorly designed class is difficult to use, making the code using it hard to write, clumsy, buggy, and hard to understand, maintain, and extend.
- A well designed class is easy to use, does the right thing, and the code using it is easy to write, maintain, and extend.
- The harder a class is to use, the more expensive it is to build and maintain the client code that uses the class.

It makes sense to expand the public interface if the additional effort expended by the developer of the class will be made up by reduced effort by developers of clients. The class developer who has a deeper understanding of what's happening in the class can do the work once correctly. The saves work that would otherwise have to be done many times, possibly incorrectly, by different developers who are building clients of the class.

However, it makes no sense to add convenience functions that won't be used by the class' clients.

This means that a good job of interface design requires extensive knowledge of how the class will be used. If you are implementing a class that is like an already existing class, you can acquire this knowledge by looking at the use of the already existing class. If you are implementing a new class that is not like an existing one, you should expect that your interface will have “bugs” that will need to be fixed.

Private Implementation

The implementation of the class is private to the class. Implementation details should not be visible to the users of the class. This is not a heuristic, it's just good design: hiding any information that the clients don't need to know.

Be wary of implementation details that “bleed through” the interface. For example, what is the best way to sort a list? If the list is backed by an array, a merge sort, for example, would be a good choice. However, for a linked list, any algorithm that sorts the list “in place” will probably have $O(n^2)$ performance rather than O(n lg n) performance. This performance issue could create a situation where the client code needs to know what kind of list a list is. (Note: Java solves this problem by sorting the list in place for ArrayLists and copying the list to an array, sorting the array, and copying the list results back into the list for LinkedLists. This way, sorting a list is always O(n lg n).)

Use Descriptive Names

Develop a convention for naming the methods, attributes, etc. of your classes. Follow your convention and use it to provide descriptive names. When you extend a class, make sure to follow the conventions in that class, even if you are modifying someone else's class.

Example: Suppose you are building a new kind of collection (vector, linked list, ...). How do you name your methods:

- add, remove, ...
- addItem, removeItem, ...
- itemAdd, itemRemove, ...
- ...

If you already have collection types in your system, you should follow the conventions that are already in place. In general, it's a good idea to imitate the naming style in any software you're modifying.

Heuristic: A Class should be Responsible for Itself

All classes should be responsible for their own operations. Don't have the client check the properties of an object and then make a decision about what can or cannot be done with the object. The object itself should be responsible for determining what can and cannot be done with the object.

A simple example:

Consider an Account object that maintains information about a checking account. We might want to know if the account is overdrawn. We could check for a negative balance. However, an account with overdraft protection might not be considered overdrawn even with a negative balance. Rather than implementing this complex set of rules in a client, the account object should provide a boolean method that answers the question. The object is now in charge of deciding if it is overdrawn.

A second example:

In the game example, a draw method can be applied to anything that is Drawable (presumably an interface):

```
for (Drawable d : allDrawables) {
    d.draw(...);
}
```

Note that no check to determine the kind of drawable is required. The method dispatching machinery will invoke the correct draw method for each object based on the actual class of the instance `d` of `Drawable`. Each `Drawable` is responsible for doing the "right thing" for itself. This means that different `Drawable`s can employ distinct techniques, e.g. an image, boxes and lines, multiple images with boxes and lines, etc., for drawing itself.

An extension of the above example:

Suppose I have a list of "interesting objects", which implement the `InterestingObject` interface. Some `InterestingObject`s are `Drawable`, meaning they implement the `Drawable` interface. `InterestingObject`s which are not `Drawable` are drawn based on a `DrawingType` which is an enumeration. (This example is a little contrived.) Consider the following code:

```
for (InterestingObject o : allInterestingObjects) {
    // Check to see if this object is a Drawable
    if (o instanceof Drawable) {
        // Call draw method on Drawable
        ((Drawable)o).draw();
    } else {
        // Call standard draw method base on DrawingType
        standardDraw(o.getDrawingType());
    }
}
```

In this case the object is not responsible for determining how it should be drawn. Here's an alternative approach: add an `isDrawable()` method to `InterestingObject` that returns true if the object is `Drawable`. Now the code looks like:

```
for (InterestingObject o : allInterestingObjects) {
    // Get Drawable
    if (o.isDrawable()) {
        // Call draw method on Drawable
```

```
              ((Drawable)o).draw();
          } else {
              // Call standard draw method base on DrawingType
              standardDraw(o.getDrawingType());
          }
      }
```

The object is now responsible for determining whether it should be drawn as a `Drawable` or drawn using the `standardDraw` method. This gives some additional flexibility. An object could choose to use the `standardDraw` method even if it also implemented the `Drawable` interface.

Here's an alternative approach: rather than requiring the object to implement `Drawable` directly, add a `getDrawable()` method to `InterestingObject` that returns a `Drawable` or `null` if the object does not have an associated `Drawable`. Now the code looks like:

```
      for (InterestingObject o : allInterestingObjects) {
          // Get Drawable
          Drawable d = o.getDrawable();
          if (d != null) {
              // Call draw method on Drawable
              d.draw();
          } else {
              // Call standard draw method base on DrawingType
              standardDraw(o.getDrawingType());
          }
      }
```

Now the object is responsible for determining how it should be drawn, not the client code. This second version of the code allows the method `o.getDrawable()` to return some other object, not necessarily just `(Drawable)o` which is what the first version must use.

This leads to the following idea: we change our objects so that `getDrawable()` always returns a `Drawable` and for objects using the `standardDraw`, the `Drawable` that's returned calls the `standardDraw` method. Now the code looks like:

```
      for (InterestingObject o : allInterestingObjects) {
           // draw object
           o.getDrawable().draw();
      }
```

Now the object is responsible for determining how it should be drawn. Note that this last version also has the advantage that `getDrawable` always returns a non-null value. So the caller does not have to worry about checking for a `null` return.

Heuristic: Keep the Scope as Small as Possible

Try to keep the scope of everything as small or restricted as possible.

Here are some versions of this rule:

- Don't create an instance variable in a class when a local variable in a method will suffice.
- In general attributes in a class should be `private`. However, if they're not `private`, they should not be `public` if package-private or `protected` will suffice.
- Make attributes `final` or constant if they are not supposed to be changed. This reduces the

scope for modification of the variable. (Local variables and parameters can also be `final` in Java.)

- Don't make methods `public` when `private`, package-private, or `protected` will work.
- Use packages and name spaces to isolate components that are internal to a sub-system.
- Don't use `static` attributes when non-static will work.
- Prefer that global variables, that is, `public static` attributes, be `final`. That is prefer global constants to global variables. Variables that are global are subject to modification by any code anywhere in the program. This is a form of coupling that should be avoided.

Heuristic: Keep Coupling Low

If a class depends on another, that is said to be *coupled* to the one it depends on. Some level of coupling is always present in software. However, high levels of coupling creates problems when software has to be changed to fix bugs or add new features.

Using interfaces rather than actual classes helps to reduce coupling. Client code is only coupled to the interface, not to the implementation. Changing the implementation of a class which implements an interface does not impact clients of the interface.

Similarly, using factory functions (see below under Constructors) to create objects helps to lower coupling by hiding the details of object construction. The code calling the factory function is only coupled to the factory function and not to the actual implementing class.

There are many additional ways to reduce coupling that we will discuss later.

Heuristic: Use Interfaces

Many object-oriented languages provide a formal concept of an interface. The concept of an interface is that it represents an Abstract Data Type (ADT), a “pure” behavior. That is, it defines a set of method signatures that constitute the interface. In general, interfaces shouldn't include implementation: no method bodies, no instance variables, no constructors. Java and Ada have included interfaces explicitly in the language. You can get the effect of an interface in C++: define an abstract class with all public abstract methods, no attributes, no constructors, and a virtual destructor.

In a programming language like Java, this leads to the following specific ideas:

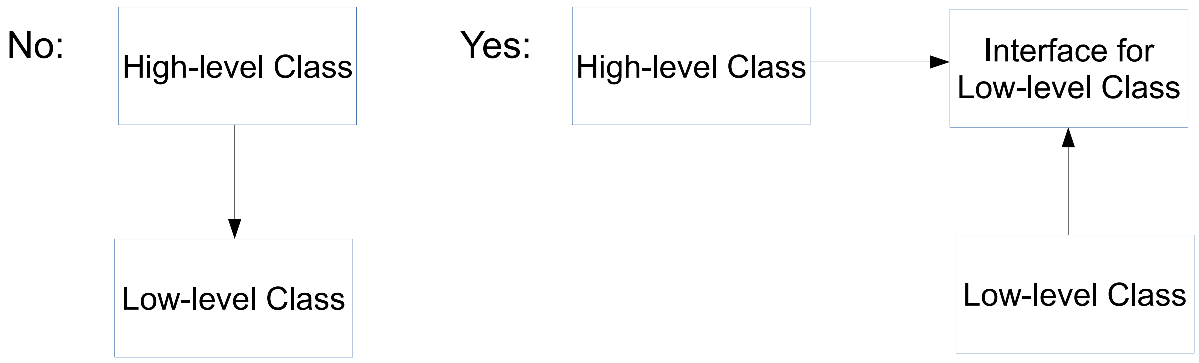
- Prefer that the type of a parameter to a method be an interface, not a class.
- Prefer that the type returned by a method be an interface, not a class.
- Only import interfaces unless you must import concrete classes.
  - In Java, `new` requires a concrete class, requiring that the concrete class be imported.

Heuristic—Dependency Inversion Principle: Depend on Interfaces not on Implementations (Classes)

The idea here is that both low-level classes and high-level classes, which depend on each other, should in fact only depend on the interfaces provided by those other classes, not their implementations.

The reason this is called dependency inversion is that traditionally, higher layer classes depend on concrete lower layer classes while lower layer classes depend on interfaces. This principle is arguing specifically that higher layer classes should also depend on interfaces.

A picture of this idea:



The primary benefit of this principle is that it opens up the possibility of multiple implementations of lower layers being used by higher layers.

Here are some specific guidelines for implementing the dependency inversion principle:

- The type of all attributes of a class should be an interface or an abstract class.
- Concrete classes should only extend an abstract class or an interface. Concrete classes should not extend another concrete class.
- All object creation must be done with some creational pattern and/or using *dependency injection*[2] that decouples object creation from the actual class of the object being created. In Java, specifically, this avoids the problem with `new` requiring a concrete class.

Heuristic—Open/Closed Principle: *Classes should be should be open for extension but closed for modification*

Bertrand Meyer[3] is generally credited with originating the Open/Closed Principle. The idea was that once completed, the implementation of a class could only be modified to correct errors; new or changed features should be implemented in a different class. That new class could reuse code from the original class through inheritance.

Later, the object-oriented community morphed the concept to make use of interfaces. In this case, all implementations inherit from an interface. You can change implementations as long as you don't change the interface.

If you create a new version of the implementation that changes behavior, you create a new, distinct interface. The new implementation has the option to implement both the old and the new interface (either directly or by having the new interface extend the old one). This means that code that wants the new features can use the new interface. Code that uses, or implements the old interface does not have to change.

Heuristic—Interface Segregation Principle: *Don’t force clients to depend on methods they don’t use*

Big interfaces—ones with many methods—are problematic. There are two potential problems with large interfaces:

[2] http://en.wikipedia.org/wiki/Dependency_injection
[3] http://en.wikipedia.org/wiki/Bertrand_Meyer

1. Typically clients will only depend on some subset of the interface—some subset of the behavior —rather than the whole interface. A change to one part of the interface may impact clients that depend on other, unrelated parts of the interface. These clients should be unaffected but may not be.
2. Classes that implement the interface are required to implement the entire interface, not just some interesting subset of the interface.

The Interface Segregation Principle tells us to divide the interface into pieces based on those parts of the interface that individual clients depend on. If you have a class that implements multiple interfaces you can handle that by the declaration:

```
class C implements I1, I2, I3, … { … }
```

If you have clients that depend on objects that implement more than one part of the interface, you can declare:

```
interface I12 extends I1, I2 { }
```

This yields an interface that is simply `I1` and `I2` combined.

Heuristic: When Possible, Prefer Value Types

In Object-Oriented programming a v*alue type* is a type that represents a value with no additional behavior. Some authors refer to value types as *immutable* types.

In Java, all primitive types are value types. However, Java has reference types, such as Integer, BigInteger, and String, that are also value types. Value types are always immutable—that is the value cannot be changed or mutated once the object is created. When you “modify” a value type, what you actually do is replace the old value with a different value. In particular, in Java, modifying a String results in a new String object. The old object remains unchanged. Similarly, if you add one BigInteger to another, the result is a new BigInteger containing the sum.

Effective Java[4], item 15, gives the following rules for immutable classes:

1. Don’t provide methods that modify the object’s state.
2. Ensure that the class can’t be extended. Generally, by making the class final. (An extension can modify the behavior, potentially destroying immutability.)
3. Make all fields final. This means that all fields must be initialized by initializers or by a constructor.
4. Make all fields private and use getters for “public” attributes. Exception: you can use public final fields of primitive or other value types. But, only do this if you are prepared to disallow future implementation changes that would require changing the attributes.
5. Ensure exclusive access to any mutable components. Don’t provide getters for mutable attributes or have the getter make a *defensive copy* of any mutable attribute before returning it.

Note: One alternative for #2 is to provide the class with a private constructor and a public static method that actually constructs objects.

[4] Effective Java (Second Edition), Joshua Bloch, 2008, Addison-Wesley. This book is also available via the WWU library Safari subscription.

Value types are useful in many different situations. In particular, an object can return an instance of a value type without having to make a defensive copy in case the calling program were to modify the value.

Effective Java gives the following benefits of immutable objects:

1. Simplicity. The internal state cannot change.
2. Thread safe. No synchronization required.
3. Can be shared freely. You can reuse instances arbitrarily. In a garbage collected language like Java, no tracking is required for memory management.
4. You can share internal data among multiple immutable instances.
5. Immutable objects are good building blocks for other objects.

Effective Java also gives the following disadvantage: A separate object is required for every distinct value. If this becomes a problem, the following solution is proposed: a mutable companion class. For example, Java has a `StringBuilder` class that can be used to construct `String`s using mutations. The `toString` method of `StringBuilder` is used to get an ordinary `String` when that is desired.

Provide Standard Operations

There are many "standard" operations that can be provided on classes. Here are some examples from Java:

- `String toString()`: Convert an object to some printable form so that it can be displayed.
- `Boolean equals(Object obj)`: Equality comparison for instances.
- `int hashCode()`: Compute a hash code for an object. If two objects are equal in the sense that the equals method returns true, those two objects should have the same hash code.
- Create a copy of an instance.
- Iterator for a collection. (Iterators are objects that allow you to loop through all the elements of a collection.)

Some notes:

- In Java all methods on the class `Object` have default implementations that can be inherited by any other object.
- `equals` and `hashCode` generally are changed together. You must change them together if you ever plan to use the class for the key to a hash table. You can get away with redefining equals without redefining hashCode if you never use the class as a key for a hash table.
- The `Object` class in Java has a `clone` method that is intended to be used to copy objects. However, it is difficult to use and get right. Effective Java, item 11, recommends staying away from the clone operation because it's very tricky to get it right. (Read the article for full details.) It suggests that you create a "copy constructor" (a constructor that takes a single argument of the type and constructs a new instance which is a copy) instead.

A Note on == and equals() in Java

In the Java language the == operator returns true, if

- For primitive types, if the two values are equal

- For reference types, if the two references reference the same object.

The `equals` method is provided as a way to test if two objects are “the same” in some sense that is specific to the object. For example, two Dates are the same if they represent the same date, and two Strings are “the same” if they have the same sequence of characters.

The default method for equals in Object simply does a == comparison. So, it’s always safe to use the equals method.

For value types you almost always need to override the default equals operator.

For any type, here’s how to correctly override the equals operator:

```
@Override public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;
    // Do whatever comparisons are needed to
    // determine if this and mt are "the same".
}
```

Note: `(null instanceof MyType)` returns false.

Design Robust Constructors

As we've discussed, *constructors* are “methods” that are responsible for creation of instances of a class. There are two variations of constructors:

1. Constructors. This is the code that gets invoked (in Java, C++, Python) when you do `object = new Class(...);` This is what is called a constructor in a programming language.
2. Factory functions. This is a function, frequently static, that (maybe) invokes a constructor (as in definition 1) and returns the desired object to the user. We will discuss factory functions in more detail later.

First and foremost, constructors (either definition) should ensure that the constructed or returned object is in a safe and stable state. Fields should be initialized to a safe, consistent, set of values. For lists, you probably want to initialize to an empty list (or table or array). In a class for rational numbers, you would want to initialize the number to the rational number zero which would have a numerator of zero and a denominator of one.

Chaining Constructors

In general, when you are constructing a subclass, you need to be able to invoke a constructor on a super class. This is called “chaining constructors”. In Java this is accomplished by using `super(...)` in the constructor of a subclass. There are other ways of doing this in languages like C++, Ada, and Python.

Warning: Consider the following Java:

```
class C1 {
    C1(...) {
        doSomething();
    }

    void doSomething() {...}
}
```

```
    class C2 extends C1 {
        C2(...) {
        }
        @Override doSomething() {...}
    }
```

Which `doSomething` method is called when a new C2 is constructed?

Answer: In Java and Python, `doSomething` in C2 is called, in C++, `doSomething` in C1 is called.

The Java approach probably has undesirable results. For example, in Java, object initialization occurs in two stages: (1) attribute initializers and initializer blocks, and (2) the constructor.

In the case above, the statement `new C2()` will result in initialization occurring in the following order:

1. Attribute initializers and initializer blocks for C1
2. The constructor for C1
3. Attribute initializers and initializer blocks for C2
4. The constructor for C2

Attribute initializers and initialization code for C2 will not have been executed when the overriding version of C2 is called. The result is that doSomething in C2 is operating on an uninitialized instance of C2.

Rule: don't call any method that can be overridden in a constructor. It is unlikely that the method will do what you expect. Calling private methods, static methods, or, in Java, final methods are all acceptable.

Heuristic: Consider using Factory Functions (Methods)

A factory function (in Java a `public static` factory method) is simply a function which is responsible for returning an instance of a class. (Note that the Factory Method design pattern is different but related.) Factory functions can be used instead of or in addition to the usual constructors. Effective Java, item 1, lists the following potential advantages of factory functions:

1. Factory functions have names. This can enhance readability. This can be used to provide multiple "constructors" with the same signature.
2. Factory functions do not have to return a new object. For example, you can cache objects that will be repeatedly constructed.
3. A factory function can return a subtype of it's return type. It can pick the appropriate actual type based on the parameters provided. This includes returning an instance of an interface.

Some potential disadvantages of factory functions:

1. Classes with only private constructors cannot be sub-classed. This can be an advantage or a disadvantage. You can allow sub-classing by providing protected or package-private constructors.
2. Static factory functions are not otherwise distinguishable from any other static function.

Effective Java recommends that you consider static factory functions as an alternative to public constructors for your classes.

Destructors

In C++, but not Java, there is a special method called a *destructor*. The destructor is called when the object's memory is freed by use of a `delete` statement or when it goes out of scope. Destructors should take whatever action is needed to cleanup any memory or other resources that are part of the object.

Because Java uses garbage collection, the most common need for destructors—memory cleanup—is eliminated. Java does have *finalizers* that are invoked when an object is to be garbage collected. These can be used to release any external resources that might be associated with the object. (Note: Effective Java recommends that you avoid finalizers if at all possible. The JRE makes no guarantees as to when finalizers are run, if ever.)

Error Handling

What do you do if your program finds an error? Here are the standard choices:

1. Ignore the problem.
2. Check for problems and abort the program when you find one.
3. Check for problems, catch the mistake, and attempt to fix the problem.
4. Throw an exception when you find a problem.

Here are some comments on these solution.

Ignore the problem

This is an all around bad idea. This is sometimes referred to as a *silent error*. The best thing that can happen in this case is that the user is confused and has to do more work to figure out what's going on. Don't do this.

Abort the Program

This allows you to display a message that, hopefully, explains the problem. This approach does allow the program to clean things up and exit gracefully. But, otherwise, this is usually not a satisfactory approach since the user usually has to restart the program and get back to where they were in their work flow.

Attempt to Recover

This is a good idea if you know what to do and when to do it. If you are trying to pop the next entry off a stack and the stack is empty, you have the option of returning an indicator that says that the stack was empty. This is a good solution assuming that the caller is prepared to check for the indicator.

As an example, the Deque class in the Java standard library has both peekFirst and getFirst methods that return the first element of the deque. These operations perform the same function except that peekFirst will return null when the deque is empty and getFirst will throw an exception.

But, what about trying to divide by zero. You can certainly change the denominator to fix the problem. But, what is the correct value? Will that result be the result that was really intended? How does the caller know that whether the result of the operation was the result of an ordinary division or the result of a division by zero.

Note (aside): Interestingly enough, the IEEE standard for floating point includes special values for +∞, -∞, and Not-a-Number (NaN) and processes these values correctly as part of computations. (For example, +∞ + 1 = +∞.) This allows a program to compute with these values and then check to see if there was a problem with the computation. The IEEE standard gives you the option to either proceed with the computation using these special values or to throw an exception. The Java standard does not give you this option, the computation proceeds and you must check for special values that might represent an error.

Heuristic: Prefer to Throw an Exception

This is generally the preferred approach unless it is obvious how to recover from the error. The exception handling mechanism provides an easy way to do cleanup as the stack is unwound.

Throwing an exception is also the preferred approach even when you think you have a programming error. (Generally, there is no good strategy for recovering from programming errors other than fixing the program.) If there is no handler for the exception, the program will terminate, which is probably what you want.

Note that, in general, exceptions should be exceptional. In many languages, but not all, handling exceptions is expensive. (In general, Java and C++ are among the languages where exceptions are expensive.) So, using exceptions for normal processing is probably not the right approach.

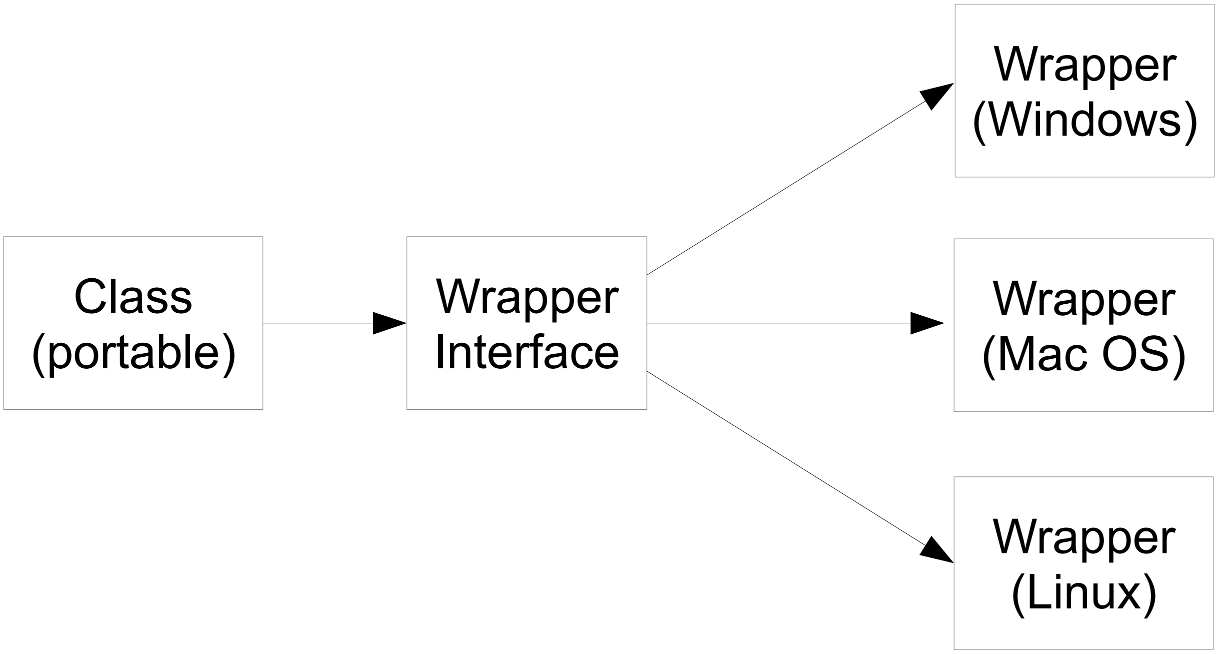
Documenting Error Handling

How the class handles errors is part of the public interface of the class. When you are designing, you need to decide how errors will be handled. For example, what will be the effect of try to pop an item off an empty stack. Both returning an indicator and throwing an exception are valid choices.

Classes are built to work with other classes. When documenting a class' behavior, you need to include information about what exceptions might be thrown in what circumstances.

Heuristic: Abstract Out Non-portable Code

When you have to write code that is non-portable, for example is dependent on the operating system, the way to do it is to isolate the code and have that code provide an interface that doesn't change between systems. You can change systems by simply changing the “wrapper” class. Here's a picture:


Class
(portable)
Wrapper
Interface
Wrapper
(Windows)
Wrapper
(Mac OS)
Wrapper
(Linux)