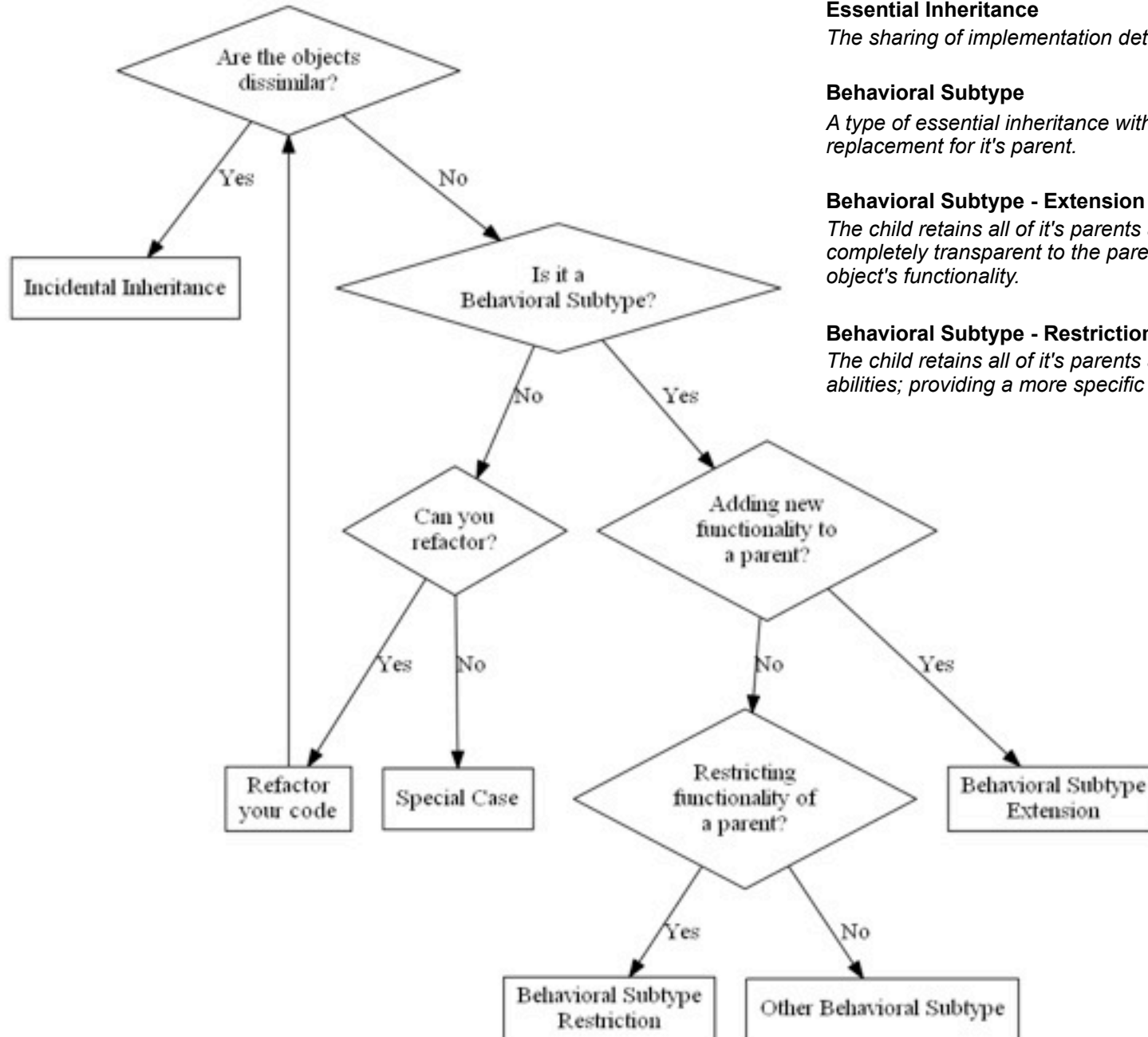


# Flowchart

Use this flowchart to help you figure out what type of inheritance you're working with.



# Definitions

## Incidental Inheritance

*The sharing of implementation details between dissimilar objects.*

## Essential Inheritance

*The sharing of implementation details between similar objects.*

## Behavioral Subtype

*A type of essential inheritance with the ability for a child to serve as a drop-in replacement for it's parent.*

## Behavioral Subtype - Extension

*The child retains all of it's parents abilities while adding new functionality that is completely transparent to the parent object, or is defined in terms of the parent object's functionality.*

## Behavioral Subtype - Restriction

*The child retains all of it's parents abilities, but restricts some (or all) of those abilities; providing a more specific implementation.*

Suppose we have a *Report* object which needs to iterate over a collection of numerical data and perform some computations, including a sum of all the values in the collection. There are several different ways we can model this:

1) We could create a *StatisticalArray* subclass and an instance of this class in our *Report* object to wrap the collection and do the computations:

```
class StatisticalArray < Array
  def sum
    reduce(:+)
  end

  # ... other stats methods
end
```

2) We could add an *each()* method to our *Report* object and mix in the *Enumerable* module, and then mix in our own *Statistics* module

```
module Statistics
  def sum
    reduce(:+)
  end

  # ... other stats methods
end
```

3) We can use simple aggregation to wrap the collection and do the necessary computations:

```
class StatisticalCollection
  def initialize(data)
```

# Incidental Inheritance

Code sharing between dissimilar objects

## Analysis and Recommendations

While approach #1 and #2 are commonly seen in Ruby programs, they have significant disadvantages to #3:

-When we take approach #1, we have to be aware of the entire API of *Array* (both public and private) in order to safely extend it, and the end result is our object is much more focused on being *Array*-like than it is on statistical computations.

-When we take approach #2, we need to treat our *Report* as if it is a collection rather than as if it simply *has a collection* it operates on, which is a violation of the single responsibility principle.

-In approach #3, we avoid both issues and maintain a very narrow surface at the cost writing slightly more verbose code.

*This is a common pattern in Ruby programs, and often, the benefits of **simple aggregation** outweigh the convenience of inheritance or module mixins when objects are dissimilar.*

*Suppose we have an XMLBuilder class and we'd like to create a subtype of this class that allows for easier building of html.*

# Behavioral Subtype - Extension

Transparently adding new functionality to a parent object

We'd start with the XMLBuilder; something as shown below

```
class XMLBuilder
  def self.build(filename, &block)
    # create new XMLBuilder, instance_eval and save
  end

  def initialize
    self.xml = []
  end

  def tag(name, attrs = {}, &block)
    # convert tag into xml and render block via
    # instance_eval
  end

  # other methods such as #to_s, etc.
end
```

Now that we have our parent class, we can create the HTMLBuilder subtype as shown below.

```
class HTMLBuilder < XMLBuilder
  def a(attrs = {}, &block)
    tag("a", attrs, &block)
  end
end
```

## Analysis and Recommendations

- We are not overwriting any existing functionality
- Child should be a drop-in replacement for parent

*Suppose we have a simple Array class that allows insertion and deletion of arbitrary objects. We can use this Array definition and restrict it's functionality to create both a Stack and a Queue.*

First we start with a definition of a SimpleArray.

```
class SimpleArray
  def initialize
    @items = []
  end

  # #insert/#delete methods that allow you to insert/delete
  # objects at any position in the array.
end
```

Given our definition of a SimpleArray, we can define a Queue by restricting #insert to inserting at the tail of the array and #delete working on the head.

```
class Queue
  # same #initialize as SimpleArray

  def insert(obj)
    @items.insert(-1, obj)
  end

  def delete
    @items.delete_at(0)
  end
end
```

# Behavioral Subtype - Restriction

Restrict capabilities of a parent object

## Analysis and Recommendations

-A restriction should be a drop in replacement. However, it will not display the same functionality, like an extension does.

-Notice you cannot write a Queue as a restriction on Stack (or vice versa). This is because their delete methods would invert the functionality of the other versus restricting it.

-You should be able to define a restriction in terms of it's parent. For example.

1. A simple array has a list of items.
2. A simple array has an insert operation to add an object at any position.
3. A simple array has a delete operation to remove an object at any position.

1. A queue is a simple array.
2. A queue's insert operation can only add an object to the tail.
3. A queue's delete operation can only remove an object from the head.

*As an example of a special case, suppose we have our Queue implementation from the “Behavioral Subtype - Restriction” section. We want to Persist this Queue on disk by creating an extension of it called PersistentQueue.*

First we start with a definition of a Queue.

1. A queue is a simple array.
2. A queue's insert operation can only add an object to the tail.
3. A queue's delete operation can only remove an object from the head.

```
class Queue
  # same as “Behavioral Subtype - Restriction” section
end
```

We now define our PersistentQueue.

1. A persistent queue is a queue.
2. A persistent queue's initialize method take a filename to persist to.
3. A persistent queue's insert method also updates the saved queue.
4. A persistent queue's delete method also updates the saved queue.

```
class PersistentQueue
  def initialize(filename)
    @filename = filename
    File.open(@filename) do |f|
      @items = Marshal.load(f.read)
    end
  end

  # #insert/#delete have the same implementation as before
  # only they call #persist before returning
end
```

# Refactoring & Special Cases

## Analysis and Recommendations

Given a first glance at PersistentQueue we might think that it's an extension. However, because it is using `Marshal` it breaks the functionality of a Queue being able to accept any object. Since Marshal doesn't serialize all Ruby objects, it will fail to load/persist all objects it's given. Because of the need to persist these Ruby objects, we cannot refactor. It would be acceptable to keep this as a special case.

- Start by trying to refactor your code.
- Do your best.