Abstraction

November 13, 2019

https://en.wikipedia.org/wiki/Abstract_nonsense

Software Engineers and Patterns

Software Engineers and Patterns

- Software engineers love architecture principles.

Software Engineers and Patterns

- Software engineers love architecture principles.
- Who has heard of *design patterns* and the gang of four book?

Software Engineers and Patterns

- Software engineers love architecture principles.
- Who has heard of *design patterns* and the gang of four book?
- The authors were actually inspired by a book that came up with a pattern language in architecture

Software Engineers and Patterns

- Software engineers love architecture principles.
- Who has heard of *design patterns* and the gang of four book?
- The authors were actually inspired by a book that came up with a pattern language in architecture
- Such patterns were meant to provide guidelines across buildings in multiple style *abstracting* away concrete details of individual buildings.

Software Engineers and Patterns

- Software engineers love architecture principles.
- Who has heard of *design patterns* and the gang of four book?
- The authors were actually inspired by a book that came up with a pattern language in architecture
- Such patterns were meant to provide guidelines across buildings in multiple style *abstracting* away concrete details of individual buildings.
- Software design patterns are thus all about providing *abstractions*.

Theories of Abstraction

Let's talk about abstraction.

Theories of Abstraction

Let's talk about abstraction.

- 

Theories of Abstraction

Let's talk about abstraction.

- 
- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.

Theories of Abstraction

Let's talk about abstraction.

- 
- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.
- Oddly enough there's a bit of a stigma against it, despite the fact that music without lyrics has far less stigma.

Theories of Abstraction

Let's talk about abstraction.

- 
- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.
- Oddly enough there's a bit of a stigma against it, despite the fact that music without lyrics has far less stigma.
- Abstraction in mathematics and computer science is about *generalization*. Take away the concrete details of certain objects and see how they are similar.

Why Abstract?

This can lead us to *classify* different objects into related groups.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.
- Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.
- Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).
- But we will study abstraction from a less mathematical viewpoint.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.
- Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).
- But we will study abstraction from a less mathematical viewpoint.
- I will show instances of *almost identical* code and how a programming language feature allows the two pieces of code to be generalized.

Buddy Functions

Let's consider the following functions in Java:

```java
public boolean hasSmith(List<String> names) {
  for (String name : names) {
    if (name == "Smith")
      return true;
  }
  return false;
}

public  boolean hasBob(List<String> names) {
  for (String name : names) {
    if (name == "Bob")
      return true;
  }
  return false;
}
```

Buddy Functions

Let's consider the following functions in Java:

```
public boolean hasSmith(List<String> names) {
  for (String name : names) {
    if (name == "Smith")
      return true;
  }
  return false;
}

public  boolean hasBob(List<String> names) {
  for (String name : names) {
    if (name == "Bob")
      return true;
  }
  return false;
}
```

These look pretty similar, right?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

- Why only design a function that searches for names?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

- Why only design a function that searches for names?
- Why not search for arbitrary items so long as they are comparable?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean hasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

- Why only design a function that searches for names?
- Why not search for arbitrary items so long as they are comparable?
- Then searching for names becomes an instance of a more general problem that is solved.

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
  boolean hasItem(T searchItem, List<T> items) {
  for (T item : items) {
    if (item.equals(searchItem))
      return true;
  }
  return false;
}
```

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
  boolean hasItem(T searchItem, List<T> items) {
  for (T item : items) {
    if (item.equals(searchItem))
      return true;
  }
  return false;
}
```

- Alright, now we can search for arbitrary comparable items!

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
  boolean hasItem(T searchItem, List<T> items) {
  for (T item : items) {
    if (item.equals(searchItem))
      return true;
  }
  return false;
}
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>>
  boolean hasItem(T searchItem, List<T> items) {
  for (T item : items) {
    if (item.equals(searchItem))
      return true;
  }
  return false;
}
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!
- Can we generalize any more?

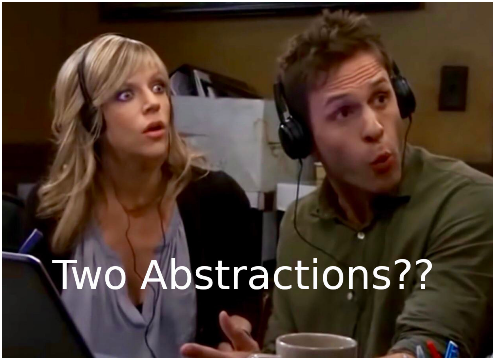
Generic Structure

So, let's use some Java *generics* to generalize our program.

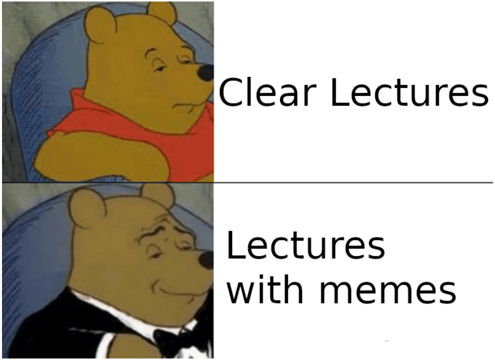
```
public <T extends Comparable<T>>
  boolean hasItem(T searchItem, List<T> items) {
  for (T item : items) {
    if (item.equals(searchItem))
      return true;
  }
  return false;
}
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!
- Can we generalize any more?
- We actually have 2 more abstractions that we can apply!

Two Abstractions??



I'm Sorry



Back on Track

Ok, let’s get down to business.

Back on Track

Ok, let's get down to business.

- 

Back on Track

Ok, let's get down to business.

- 

- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.

Back on Track

Ok, let's get down to business.

- 

- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.
- So if we wanted to check if a string equaled smith we could write:

```
Predicate<String> isSmith = str -> str == "Smith";
```

Back on Track

Ok, let's get down to business.

- 

- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.
- So if we wanted to check if a string equaled smith we could write:
`Predicate<String> isSmith = str -> str == "Smith";`
- I could then pass this as the first argument to a funciton `TestItems` that I will now define.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
  boolean testItems(Predicate<T> pred, List<T> items) {
  for (T item : items) {
    if (pred.test(searchItem))
      return true;
  }
  return false;
}
```

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
  boolean testItems(Predicate<T> pred, List<T> items) {
  for (T item : items) {
    if (pred.test(searchItem))
      return true;
  }
  return false;
}
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
  boolean testItems(Predicate<T> pred, List<T> items) {
  for (T item : items) {
    if (pred.test(searchItem))
      return true;
  }
  return false;
}
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.
- For example, if I define iterators over dictionaries or trees we should still be able to apply a predicate to them.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>>
  boolean testItems(Predicate<T> pred, List<T> items) {
  for (T item : items) {
    if (pred.test(searchItem))
      return true;
  }
  return false;
}
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.
- For example, if I define iterators over dictionaries or trees we should still be able to apply a predicate to them.

```
public <T extends Comparable<T>>
  boolean testItems(Predicate<T> pred, Iterable<T> items) {
    for (T item : items) {
      if (pred.test(searchItem))
        return true;
    }
    return false;
```

Predicates as Arguments

Our third abstraction was a bit strange, right?

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.
- It then applied this function to every element in the list and observed whether it returned `true` for that element.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.
- It then applied this function to every element in the list and observed whether it returned `true` for that element.
- This is a very powerful concept where we can determine if perhaps all elements in a collection satisfy a property or even one element. Or we can collect all individuals in a collection that satisfy some property.

Predicates as Arguments

Our third abstraction was a bit strange, right?

- I used this weird predicate type in Java and then assigned an *anonymous function*.
- This is a function with no name, and it is useful if you only need to use a function in one place in your program.
- The second thing is that we had a parameter that received a function as input.
- It then applied this function to every element in the list and observed whether it returned `true` for that element.
- This is a very powerful concept where we can determine if perhaps all elements in a collection satisfy a property or even one element. Or we can collect all individuals in a collection that satisfy some property.
- For example, let's consider writing a program that only admits people that are 18 and older.

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?
- Via recursion over a list of course.

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?
- Via recursion over a list of course.
- I'll skip to providing a skeleton:

Guarding Entry

Let's consider that we have a list of people whose ages are represented by Natural numbers and that we want to only collect the people over 18. Assume a person has an age field that we can project.

- How do we start writing such a function?
- Via recursion over a list of course.
- I'll skip to providing a skeleton:

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
       (... (first patrons)) ... (bar-entry (rest patrons))]))
```

-

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.
- How do I get the age out?

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.
- How do I get the age out?
- `(person-age (first patrons))`

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.
- How do I get the age out?
- `(person-age (first patrons))`
- How do I check whether the age is less than 18?

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.
- How do I get the age out?
- `(person-age (first patrons))`
- How do I check whether the age is less than 18?
- `(< (person-age (first patrons)) 18)`

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.
- How do I get the age out?
- `(person-age (first patrons))`
- How do I check whether the age is less than 18?
- `(< (person-age (first patrons)) 18)`
- Thus, we should get `(< 26 18)` since we are getting Sara as our person, and this should return false.

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.
- How do I get the age out?
- `(person-age (first patrons))`
- How do I check whether the age is less than 18?
- `(< (person-age (first patrons)) 18)`
- Thus, we should get `(< 26 18)` since we are getting Sara as our person, and this should return false.
- Since this returns false, we should throw Sara into the list we're building. How do we do that?

Guarding Entry

So, in order to protect against underage patrons getting in, we must do a comparision.

- Assume that `(first patrons)` returns `(person "Sara" 26)`.
- How do I get the age out?
- `(person-age (first patrons))`
- How do I check whether the age is less than 18?
- `(< (person-age (first patrons)) 18)`
- Thus, we should get `(< 26 18)` since we are getting Sara as our person, and this should return false.
- Since this returns false, we should throw Sara into the list we're building. How do we do that?
- With `(cons (first patrons) ...)`

Only Collecting *Some* Items

But what if `(first patrons)` returns `(person "Dustin" 16)`?

Only Collecting *Some* Items

But what if `(first patrons)` returns `(person "Dustin" 16)`?

- Then, we had
  `(< (person-age (person "Dustin" 16)) 18)`

Only Collecting *Some* Items

But what if `(first patrons)` returns `(person "Dustin" 16)`?

- Then, we had
  `(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.

Only Collecting *Some* Items

But what if `(first patrons)` returns `(person "Dustin" 16)`?

- Then, we had
  `(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list. So we shouldn't use a cons operation.

Only Collecting *Some* Items

But what if `(first patrons)` returns `(person "Dustin" 16)`?

- Then, we had
  `(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list. So we shouldn't use a cons operation.
- There are two ways around this. One is to locally add a complicated if expression. The second is to design a new function that only does a cons operation if the first element of the list has an age greater than 18 and otherwise returns the sublist that already had filtered out underage people.

Only Collecting *Some* Items

But what if `(first patrons)` returns `(person "Dustin" 16)`?

- Then, we had
  `(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list. So we shouldn't use a cons operation.
- There are two ways around this. One is to locally add a complicated if expression. The second is to design a new function that only does a cons operation if the first element of the list has an age greater than 18 and otherwise returns the sublist that already had filtered out underage people.
- Let's consider designing this second function, called `cons-over-18`

Only Collecting *Some* Items

But what if `(first patrons)` returns `(person "Dustin" 16)`?

- Then, we had
  `(< (person-age (person "Dustin" 16)) 18)`
- which evaluates to `(< 16 18)` and then return true.
- This means that we must not add Dustin to our patron list. So we shouldn't use a cons operation.
- There are two ways around this. One is to locally add a complicated if expression. The second is to design a new function that only does a cons operation if the first element of the list has an age greater than 18 and otherwise returns the sublist that already had filtered out underage people.
- Let's consider designing this second function, called `cons-over-18`
- Does this function need to do recursion itself?

Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?

Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?
- We assume that the recursion for `bar-entry` already filters out people from sublists.

Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?
- We assume that the recursion for `bar-entry` already filters out people from sublists.
- So, our helper function simply takes in a person and a list and only conses the person onto the list if they are over 18.

Designing a Helper Function

Thankfully, our helper function does not need to do recursion!

- Why?
- We assume that the recursion for `bar-entry` already filters out people from sublists.
- So, our helper function simply takes in a person and a list and only conses the person onto the list if they are over 18.

```
;; List<person> -> List<person>
;; Add a person to the patron list as
;; long as they are over 18.
(define (cons-over-18 possible-patron patrons)
  (if (>= (person-age possible-patron) 18)
      (cons possible-patron patrons)
      patrons))
```

-

Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
       (cons-over-18 (first patrons) (bar-entry (rest patrons)))]))
```

-

Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
       (cons-over-18 (first patrons) (bar-entry (rest patrons)))]))
```

-
- In general, if it seems like the list you're building depends on the structure of the values in the list, then you need some kind of cons operation that checks properties on the head of the list.

Finishing Our Bouncer Program

Alright, with our helper combinator we can structure our recursive function in the usual way, instead of having to add an additional condition in the body of the function.

```
;; List<Person> -> List<Person>
;; Only allows patrons over 18 to enter the bar
(define (bar-entry patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons)
       (cons-over-18 (first patrons) (bar-entry (rest patrons)))]))
```

-
- In general, if it seems like the list you're building depends on the structure of the values in the list, then you need some kind of cons operation that checks properties on the head of the list.
- But what happens if the condition we are checking needs to change?

Adjusting Our Bouncer Program

Concretely, let's say that that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

Adjusting Our Bouncer Program

Concretely, let's say that that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

- Now our bar can't afford to give drinks to patrons under 21 years old. We decide that the easiest solution is to only allow 21 and over patrons, to prevent underage people from getting in and sneakily get drinks.

Adjusting Our Bouncer Program

Concretely, let's say that that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

- Now our bar can't afford to give drinks to patrons under 21 years old. We decide that the easiest solution is to only allow 21 and over patrons, to prevent underage people from getting in and sneakily get drinks.
- To model this behavior, our program must change `cons-over-18` to something named `cons-over-21` that changes its check to make sure patrons are at least 21.

Adjusting Our Bouncer Program

Concretely, let's say that that our bar was serving alcohol to patrons under 21 (this is illegal but common...) and a new Sheriff comes in and is stricter on enforcing alcohol laws.

- Now our bar can't afford to give drinks to patrons under 21 years old. We decide that the easiest solution is to only allow 21 and over patrons, to prevent underage people from getting in and sneakily get drinks.
- To model this behavior, our program must change `cons-over-18` to something named `cons-over-21` that changes its check to make sure patrons are at least 21.
- But let's say we still need to model another bar that allows people over 18, but charges them covers. Then we will need four functions for our program.

Repetitive Bar Program

```
(define (cons-over-18 possible-patron patrons)
  (if (>= (person-age possible-patron) 18)
      (cons possible-patron patrons)
      patrons))
(define (bar-entry-18 patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons) (cons-over-18
                       (first patrons)
                       (bar-entry-18 (rest patrons)))]))
(define (cons-over-21 possible-patron patrons)
  (if (>= (person-age possible-patron) 21)
       (cons possible-patron patrons)
       patrons))
(define (bar-entry-21 patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons) (cons-over-21
                      (first patrons)
                      (bar-entry-21 (rest patrons)))]))
```

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.
- On the other, most developers get bored of re-reading similar code.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.
- On the other, most developers get bored of re-reading similar code.
- To minimize failure locations and keep interest, we should *abstract* out patterns in the code.

How to Eliminate Redundant Code?

So, we can see we have a lot of redundant code in the program.

- Redundant code should cause developers an itch.
- On the one hand, there are more places for the program to go wrong.
- On the other, most developers get bored of re-reading similar code.
- To minimize failure locations and keep interest, we should *abstract* out patterns in the code.
- So, how do we get started?

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.
- But wouldn't it be more elegant to just have a numeric parameter that is used as the threshold for our checks?

Adding Parameters

A rule of thumb when abstracting code is that either you need another parameter or you need another function.

- Let's consider adding another parameter. We can add a boolean parameter that is a flag indicating whether to check if patrons are over 18 if the flag is false and over 21 if it is true.
- But wouldn't it be more elegant to just have a numeric parameter that is used as the threshold for our checks?

```
(define (cons-over age possible-patron patrons)
  (if (>= (person-age possible-patron) age)
      (cons possible-patron patrons)
      patrons))
(define (bar-entry age patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons) (cons-over age
                                (first patrons)
                                (bar-entry (rest patrons)))]))
```

-

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?
- Just like the abstraction I provided at the beginning of this material in Java, we can add a predicate as a parameter!

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?
- Just like the abstraction I provided at the beginning of this material in Java, we can add a predicate as a parameter!
- We then only cons items that meet this predicate!

What if We Need More Restrictions?

Now, let's say that we need to check that patrons have a valid id that hasn't expired. Assume that our persons struct has been updated with id information.

- How do we have to change to our program to account for this new restriction?
- We need to change `cons-over` to add an additional check.
- Luckily, this restriction applies to all bars, so we don't have to add an additional parameter.
- But in general, as we add restrictions, we have to keep modifying `cons-over`.
- Is there any way we can abstract over these changes?
- Just like the abstraction I provided at the beginning of this material in Java, we can add a predicate as a parameter!
- We then only cons items that meet this predicate!
- This is the power of having first class functions in your language!

Filtering Functions

You must be thinking "show me the code!" by now, so here it is:

```
(define (over-18? patron) (>= (person-age patron) 18))
(define (over-21? patron) (>= (person-age patron) 21))

(define (cons-over-p pred possible-patron patrons)
  (if (pred possible-patron)
      (cons possible-patron patrons)
      patrons))

(define (bar-entry pred patrons)
  (cond
    [(empty? patrons) patrons]
    [(cons? patrons) (cons-over-p
                       pred
                       (first patrons)
                       (bar-entry (rest patrons)))]))
```

Accidental Derivations

Accidental Derivations

- We can now have our separate bars filter out by different ages with: `(bar-entry over-18? PATRONS)` and `(bar-entry over-21? PATRONS)`

Accidental Derivations

- We can now have our separate bars filter out by different ages with: `(bar-entry over-18? PATRONS)` and `(bar-entry over-21? PATRONS)`
- It turns out however that our functions have become general to the point of being badly named.

Accidental Derivations

- We can now have our separate bars filter out by different ages with: `(bar-entry over-18? PATRONS)` and `(bar-entry over-21? PATRONS)`
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call `(bar-entry? string? '(1 "foo" (point 1 2) "bar"))` and the function call returns `'("foo" "bar")`

Accidental Derivations

- We can now have our separate bars filter out by different ages with: `(bar-entry over-18? PATRONS)` and `(bar-entry over-21? PATRONS)`
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call `(bar-entry? string? '(1 "foo" (point 1 2) "bar"))` and the function call returns `'("foo" "bar")`
- It turns out our bouncer function ended up being able to get all of the lists out of a heterogeneous list.

Accidental Derivations

- We can now have our separate bars filter out by different ages with: `(bar-entry over-18? PATRONS)` and `(bar-entry over-21? PATRONS)`
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call `(bar-entry? string? '(1 "foo" (point 1 2) "bar"))` and the function call returns `'("foo" "bar")`
- It turns out our bouncer function ended up being able to get all of the lists out of a heterogeneous list.
- It turns out that we have derived a famous function and called it bar-entry!

Accidental Derivations

- We can now have our separate bars filter out by different ages with: `(bar-entry over-18? PATRONS)` and `(bar-entry over-21? PATRONS)`
- It turns out however that our functions have become general to the point of being badly named.
- For example, I can call `(bar-entry? string? '(1 "foo" (point 1 2) "bar"))` and the function call returns `'("foo" "bar")`
- It turns out our bouncer function ended up being able to get all of the lists out of a heterogeneous list.
- It turns out that we have derived a famous function and called it bar-entry!
- The name of this function is `filter`

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

```
(filter
  (lambda (s) (>= (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

```
(filter
  (lambda (s) (>= (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

- And this returns `'("Smith" "Jenkins" "Brown")`

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

```
(filter
  (lambda (s) (>= (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

- And this returns `'("Smith" "Jenkins" "Brown")`
- You might be wondering what `(lambda (s) (>= (string-length s) 1))` is.

Hey Man Nice Shot

Ah Filter, everyone's favorite 90's 2 hit wonder!

- But anyway, filter takes in a predicate as its first argument and a list to collect items out of (when the predicate returns true on a list item, it is collected).
- For example, we can call `(filter even? '(1 2 3 4 5 6))`
- The result is `'(2 4 6)`
- We can call:

```
(filter
  (lambda (s) (>= (string-length s) 1))
  '("Smith" "Jenkins" "Brown" "Z"))
```

- And this returns `'("Smith" "Jenkins" "Brown")`
- You might be wondering what `(lambda (s) (>= (string-length s) 1))` is.
- It is an anonymous function, just like the one I previously showed in Java.

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.
- `(filter (lambda (x) (< x 5)) '(1 2 4 8 16))` provides an anonymous function (that is a predicate) as an argument to `filter`

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.
- `(filter (lambda (x) (< x 5)) '(1 2 4 8 16))` provides an anonymous function (that is a predicate) as an argument to `filter`
- We can also write functions that return functions.

Anonymity

Anonymous functions, functions as arguments, and functions as return values are powerful features in functional programming languages that have made their way into most mainstream languages.

- In Racket, we introduce an anonymous function with `lambda`
- `(lambda (x) (add1 x))` is an anonymous function with one parameter `x` that increments the number.
- We can also take functions as arguments to functions.
- `(filter (lambda (x) (< x 5)) '(1 2 4 8 16))` provides an anonymous function (that is a predicate) as an argument to `filter`
- We can also write functions that return functions.
- Here is a small example:

```
(define (add x)
  (lambda (y)
    (+ x y)))
```

Returning Functions

What do you think the following code is doing?

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))
(define add-three (add 3))
```

-

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))
(define add-three (add 3))
```

-
- Well, let's substitute 2 for x and then look at the body of the original function:

```
(lambda (y)
  (+ 2 y))
```

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))
(define add-three (add 3))
```

-
- Well, let's substitute 2 for x and then look at the body of the original function:

```
(lambda (y)
  (+ 2 y))
```

- And when we apply 3 in defining add-three we substitute 3 and have:

```
(lambda (y)
  (+ 3 y))
```

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))
(define add-three (add 3))
```

-
- Well, let's substitute 2 for x and then look at the body of the original function:
  ```
  (lambda (y)
    (+ 2 y))
  ```
- And when we apply 3 in defining add-three we substitute 3 and have:
  ```
  (lambda (y)
    (+ 3 y))
  ```
- So, in the first case we are returning an anonymous function that adds 2 to y, and in the latter case we are returning an anonymous function that adds 3 to y.

Returning Functions

What do you think the following code is doing?

```
(define add-two (add 2))
(define add-three (add 3))
```

-
- Well, let's substitute 2 for x and then look at the body of the original function:
  ```
  (lambda (y)
    (+ 2 y))
  ```
- And when we apply 3 in defining add-three we substitute 3 and have:
  ```
  (lambda (y)
    (+ 3 y))
  ```
- So, in the first case we are returning an anonymous function that adds 2 to y, and in the latter case we are returning an anonymous function that adds 3 to y.
- So, in a sense we are deanonymizing these two functions by using define with add-two and add-three.

Famous Higher Order Functions

There are a couple of famous higher order functions that you learn about in math. Namely derivation, integration, and composition.