Code reuse

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
Code duplication

Three (main) reasons to remove duplication

1. Since software changes, duplicated code changes need to be mirrored or else bugs will follow
2. Too much code obscures intent
3. Good abstractions help build more complicated code on top of it
Duplication bugs

- Any piece of code can, and will change
- In particular, if the same decision (configuration / algorithm) is made in multiple place, it will have to be changed in multiple places
  - But tracking down all infestations is hard or downright impossible
  - In other words, this will result in bugs
Obscures intent

- Too much code makes the code harder to follow
- Extracting similar code to a single location is a clear win since we’re reducing multiple pieces of code to just one
- By creating new modules, be they classes or methods or packages, we can also name them to help our understand and the understand of our clients
(Software) Engineering is built on layers of abstraction

- From the macro level
  - Programs on top of operating systems on top of hardware on top of logical circuits
- To the algorithmic level
  - Algorithms on top of complicated data structures on top of primitive data structures (arrays and pointers)
- To the software level
  - Services on top of services on top of services
Abstraction layers (cont.)

- This recursive structure is reflected in programming languages
  - Classes on top of classes, as far down as needed
  - Ending with primitives
- Without these abstractions, it would be impossible to build anything non-trivial
  - Imagine you had to write an entire application from scratch every time
- By extracting duplicated code, we make it easier to reuse and build more complicated abstractions on top of it
You’ve already learnt in Intro to CS that code duplication should be eradicated

So what’s the big deal?
- Extracting code is much harder between two modules than in a single module
  - Can’t extract a private method
- You have some bad habits to unlearn
- De-duplication is not always the right choice
There are many ways to achieve code reuse in Java, in descending order of goodness:

1. Promote to object method
2. Polymorphism
   - Subtyping
   - Generics
3. Composition
4. Inheritance
public class DirectorAverage {
    public double movieAverage(Director d) {
        return d.getMovies()
            .stream()
            .mapToDouble(m -> m.getVotes().stream()
                          .mapToInt(MovieVote::score)
                          .average().getAsDouble())
            .average().getAsDouble();
    }
}

public class BestMovie {
    public Movie bestMovie(List<Movie> movies) {
        movies.stream()
            .max(Comparator.comparing(m -> m.getVotes().stream()
                              .mapToInt(MovieVote::score)
                              .average().getAsDouble()))
            .get();
    }
}
```java
class Movie {
    public double getAverage() {
        return getVotes().stream()
            .mapToInt(MovieVote::score)
            .average().getAsDouble();
    }
}

public class DirectorAverageCalc {
    public double calculateMovieAverage(Director d) {
        return d.getMovies()
            .stream()
            .mapToDouble(Movie::getAverage)
            .average().getAsDouble();
    }
}

public class BestMovieCalculator {
    public Movie getBestMovie(List<Movie> movies) {
        movies.stream()
            .max(Comparator.comparing(Movie::getAverage))
            .get();
    }
}
```

Reuse patterns
Promote to object method (summary)

- Pros:
  - Logical place for such a method to be
  - No need for additional classes to be passed around

- Cons:
  - Can hurt **cohesion** if overused
  - Not always possible
    - For example, if you don’t control the class, as in standard library collections or primitives
    - But in other languages (C#, Kotlin, Scala) we can use some variation of extension methods
    - Although that brings with it its own share of problems...
Polymorphism

Generics or using base class parameters

- No additional classes or methods
  - And therefore, no loss of code locality
- Unbounded number of uses
  - Can use polymorphic code even in scenarios you haven’t thought of for free
- Cons?
  - Complicates call site
    - Especially once you get into wildcard hell
  - Usually not possible
public class MovieVoteCounter {
    private final List<MovieVote> votes = new ArrayList<>();
    public void addVote(MovieVote mv) {
        votes.add(mv);
    }
    public Movie getAverage() {
        return votes.stream()
            .mapToDouble(MovieVote::getScore)
            .average().getAsDouble();
    }
}

public class AlbumVoteCounter {
    private final List<AlbumVote> votes = new ArrayList<>();
    public void addVote(AlbumVote mv) {
        votes.add(mv);
    }
    public double getAverage() {
        return votes.stream()
            .mapToDouble(AlbumVote::getScore)
            .average().getAsDouble();
    }
}
public class VoteCounter {
    private final List<Vote> votes = new ArrayList<>();
    public void addVote(Vote v) {
        votes.add(v);
    }
    public double getAverage() {
        return votes.stream()
            .mapToDouble(Vote::getScore)
            .average().getAsDouble();
    }
}
Generics example

```java
public class BoxOfficeSmashers {
    private final Map<Movie, Integer> viewers = new HashMap<>();
    public void addViewer(Movie m, int numberOfViewers) {
        votes.put(m, votes.getOrDefault(m, 0) + numberOfViewers);
    }
    public Movie getMostPopularMovie() {
        return votes.stream()
            .max(Map.Entry.comparingByValue())
            .get();
    }
}

public class BillboardsToppers {
    private final Map<Album, Integer> Listeners = new HashMap<>();
    public void addListener(Album m, int numberOfListeners) {
        votes.put(m, votes.getOrDefault(m, 0) + numberOfListeners);
    }
    public Album getMostPopularAlbum() {
        return votes.stream()
            .max(Map.Entry.comparingByValue())
            .get();
    }
}
```
public class MostPopularCounter<T> {

    private final Map<T, Integer> viewers = new HashMap<>();

    public void addViewer(T m, int numberOfViewers) {
        votes.put(m, votes.getOrDefault(m, 0) + numberOfViewers);
    }

    public T getMostPopularElement() {
        return votes.stream()
            .max(Map.Entry.comparingByValue())
            .get();
    }
}

Reuse patterns

Gal Lalouche
We could have also used generics to solve the VoteCounter counter problem

- It would have involved more code
- But would have been more typesafe
  - No chance of counting album and movie votes together
  - But the extra typesafety isn’t always needed
    - E.g., if you’re only using VoteCounter internally and there is no chance for confusion
- Using subtyping over generics is also in line with Principle of Least Power
Generics and Wildcards

Unfortunately, Java has call-site variance rather than declaration variance

- But some interfaces are inherently variant!
  - You almost never want to accept an invariant Function type
  - That’s why all the method signatures in Stream look like this:

```java
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

- If you’re only iterating over a collection, it could be covariant
- They make the API more comfortable to use, at (almost) no cost to the client!
  - Almost, because reading the signatures is still an eye-sore
If you’re having trouble remembering when to use `super` and when to use `extends`, remember **PECS**

- **Producer Extends Consumer Super**
  - You’re iterating over a list? You’re accepting a `producer` (the list), so use `extends`
  - You’re adding items to a list? You’re accepting a `consumer` (the list), so use `super`

- **Don’t return wildcard types**
  - They force the caller to declare their type as a wildcard
  - Good wildcard are invisible to the user and provide needed flexibility
  - If the user has to deal with wildcards themselves, something is wrong in your API

---

1 Horrible art by Rob Liefield
Wildcards and immutability

Why are wildcards needed in the first place? (OOP Reminder)

- In other words, why doesn’t a List<A> extend a List<B> if A extends B?
- This actually happens with arrays, which are covariant in Java, and it is not a good thing
- The problem is object mutability, which we will talk about at length in the future

```java
String ss = new String[1];
Object[] os = ss;
// Runtime exception!
os[0] = 5;
```
- Luckily, generic classes (but not arrays) protect us from such type failures

```java
void demo(List<? extends String> producer,
          List<? super String> consumer) {
    String s = producer.get(0); // Fine
    String s = consumer.get(0); // Won't compile!
    consumer.set(0, "foo"); // Fine
    producer.set(0, "foo"); // Won't compile!
}
```

- Although more verbose than they should be, at least they are type-safe...

- In summary: accept wildcard parameters to make your API more flexible, but never return wildcards
  - Although for some APIs, the increased flexibility isn't worth the extra tokens
Generic simplicity

Avoid pointless generics and wildcard (Fewest elements and Principle of least power):

```java
// Completely equivalent to List<T> due to type inference
public <T> T getFirst(List<? extends T> ls) {
    return ls.iterator().next();
}

// No use of T; make it an existential (<??>) type
public <T> boolean isNonEmpty(List<T> list) {
    return !list.isEmpty();
}

// <? extends Object> is the same as <?
public long countNulls(List<? extends Object> list) {
    return list.stream().filter(Objects::isNull).count();
}
```
Using generics isn’t always possible, e.g., if your duplicated code is only a small portion of a much larger code base.

- It doesn’t make sense to make the entire class generic just to support a small piece of functionality.

The more common case is that we have two similar pieces of code that are hosted in much larger code bases.

The two main solutions to this problem are Composition and Inheritance.
public class MovieManager {
    private final Map<Movie, Integer> viewers = new HashMap<>();
    public void addViewer(Movie m, int numberOfViewers) {
        votes.put(m, votes.getOrDefault(m, 0) + numberOfViewers);
    }
    public Movie getMostPopularMovie() {
        return votes.stream()
            .max(Map.Entry.comparingByValue())
            .get();
    }

    // A whole bunch of other code
}

public class AlbumManager {
    // You get the picture...
}
Solving the problem using inheritance

```java
public class VoteManager<T> {
    private final Map<V, Integer> viewers = new HashMap<>();
    public void addViewer(T m, int numberOfViewers) {
        votes.put(m, votes.getOrDefault(m, 0) + numberOfViewers);
    }
    public T getMostPopularElement() {
        return votes.stream()
            .max(Map.Entry.comparingByValue())
            .get();
    }
}

public class MovieManager extends VoteManager<Movie> {
    // The viewers field, addViewer method, and
    // getMostPopularMovie method are removed
    // Everything else behaves the same
}

public class AlbumManager extends VoteManager<Album> {
    // Likewise
}
```
Solving the problem using composition

```java
public class VoteManager<T> {
    // Exact same code as before
}

public class MovieManager {
    private final VoteManager<Movie> vm = new VoteManager<>();
    public void addViewer(Movie m, int numberOfViewers) {
        vm.addViewer(m, numberOfViewers);
    }
    public Movie getMostPopularMovie() {
        return vm.getMostPopularElement();
    }
}

public class AlbumManager {
    private final VoteManager<Album> vm = new VoteManager<>();
    // Yada yada yada
}
```
You should, generally speaking, **prefer composition over inheritance**

<table>
<thead>
<tr>
<th>Composition</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible</td>
<td>Fixed</td>
</tr>
<tr>
<td>Multiple compositions</td>
<td>Single inheritance</td>
</tr>
<tr>
<td>Linear combinations</td>
<td>Exponential combinations</td>
</tr>
<tr>
<td>Encapsulated</td>
<td>Exposed</td>
</tr>
<tr>
<td>Controlled interface</td>
<td>Uncontrolled interface</td>
</tr>
<tr>
<td>More code to setup</td>
<td>Less code</td>
</tr>
<tr>
<td>Not polymorphic</td>
<td>Polymorphic</td>
</tr>
<tr>
<td>Decoupled</td>
<td>Coupled</td>
</tr>
</tbody>
</table>
Composition is flexible
- We can replace the composed element at run-time using dependency injection

Inheritance is fixed
- Can’t change inherited class in run-time
- Not true for all languages (e.g., prototype-based languages such as Javascript)
In Java one can only inherit one class
  Therefore if want to pick and choose different combinations, you have to create an *exponential* number of classes
  But you if you use composition, you just select different dependencies

But wait, wasn’t this fixed in Java 8 with *default* methods?
In Java one can only inherit one class

- Therefore if want to pick and choose different combinations, you have to create an **exponential** number of classes
- But you if you use composition, you just select different dependencies

But wait, wasn’t this fixed in Java 8 with **default** methods?

- Interface can’t hold state, so it only solves very specific problems
- Doesn’t solve our original problem for example
Encapsulation and control

- You can’t hide your inheritance
- Your solution is exposed for the world to see
- **Exposition breeds Coupling**
  - Clients may use the parent class
  - And now you can’t easily change what should have been an implementation detail
- The parent class may offer more functionality than we need it to
  - Since you can’t hide methods in the inheriting class, the derived class must offer them as well
  - Classic example: Stack extends Vector(?!)
- The parent class may change its interface without us being aware
The only real advantage of inheritance is reducing total code written.

- In our composition solution, we had to repeat the method definitions.
- But we got this for free when we used inheritance.

Composition usually involves quite a bit more code, often to the point of duplication.
One common (anti-)pattern is using inheritance to provide utility methods to the inheriting class.

```java
abstract class MovieOrder {
    protected void checkout(int price) { ... }
    protected int priceForTimeOfDay(Time timeOfDay) { ... }
    public abstract void order(int numberOfTickets, Time timeOfDay);
}

class MovieOrderWithServiceCharge extends MovieOrder {
    private static final int SERVICE_CHARGE = ...;
    @Override public void order(int numberOfTickets, Time timeOfDay) {
        int basePrice = priceForTimeOfDay(timeOfDay);
        int totalPrice = numberOfTickets * (basePrice + SERVICE_CHARGE);
        checkout(totalPrice);
    }
}

class MovieOrderWithBulkDiscount extends MovieOrder {
    private int discountForBulkOrder(int number) {
        // A larger order gets a bigger discount
    }
    @Override public void order(int numberOfTickets, Time timeOfDay) {
        int basePrice = priceForTimeOfDay(timeOfDay);
        int totalPrice = numberOfTickets * basePrice - discountForBulkOrder(numberOfTickets);
        checkout(totalPrice);
    }
}
```
The concrete classes are used as **configuration**

The client code can decide on implementation by creating a different class

```java
MovieOrder mo = new MovieOrderWithServiceCharge();
mo.order(5, AFTERNOON);
```

So what's the problem with this approach?
There's annoying boiler plate in invoking priceForTimeOfDay

But much worse, all concrete classes have to manually invoke checkout!

- This is worse than “just” code duplication
- New implementation are expected to repeat this code, but there's no way to enforce it
- Requirements that aren't enforced is basically begging for bugs
The solution to this problem is called the **Template pattern**

- We see that both concrete MovieOrder implementations share a common template
  1. Get base price
  2. Calculate total price
  3. Checkout

- The only unique part is calculating the total price

- We can apply **inversion of control** and move the main logic to the abstract parent class

- This would reduce duplication and make our code safer and clearer
abstract class MovieOrder {
    private void checkout(int price) { ... }
    private int priceForTimeOfDay(Time timeOfDay) { ... }
    // This is called the template method
    protected abstract int calculateTotalPrice(int basePrice, int numberOfTickets);
    public void order(int numberOfTickets, Time timeOfDay) {
        int basePrice = priceForTimeOfDay(timeOfDay);
        int totalPrice = calculateTotalPrice(basePrice, numberOfTickets);
        checkout(totalPrice);
    }
}

class MovieOrderWithServiceCharge extends MovieOrder {
    private static final int SERVICE_CHARGE = ...;
    @Override protected int calculateTotalPrice(int basePrice, int numberOfTickets) {
        return numberOfTickets * (basePrice + SERVICE_CHARGE);
    }
}

class MovieOrderWithBulkDiscount extends MovieOrder {
    private int discountForBulkOrder(int number) { ... }
    @Override protected int calculateTotalPrice(int basePrice, int numberOfTickets) {
        return numberOfTickets * basePrice - discountForBulkOrder(numberOfTickets);
    }
}
Template conclusions

- The template method suffers many of the same drawbacks as other uses of inheritance
  - In particular, an exponential number of combinations
- Of course, any template can be replaced with a composition
  - Simply pass an interface with the same abstract template methods
  - By passing an anonymous implementation (or even a lambda), you can also reduce the number of elements used, leading to less code
- Over-use of the template pattern can lead to confusing 'yo-yo' code
  - Controlled is constantly passed from parent to child and vice-versa, making the code harder to follow
- Generally speaking, class inheritance as configuration is something you should avoid
Template vs Convenience functionality

A pattern that is similar, but intrinsically different from template method is that of **convenience methods**

- For example, a List<T> has many methods, but most can be implemented using only a subset of those.
- It would be easier for implementers if all those methods would have a default implementation in the parent class.
  - Either an abstract class in pre-Java 8, cf. AbstractList<T>
  - Or with, as the name implies, **default** methods
  - Clients that wish to implement a more efficient variation can **override**
    - E.g., ArrayList<T> can implement sublist(int, int) more efficiently than LinkedList<T>
    - E.g., LinkedList<T> can implement remove(int) more efficiently than ArrayList<T>
While one could use composition to implement convenience methods (as always), there’s no real reason to

- Convenience methods are (hopefully) centered around a single role, so there’s no risk of exponential explosions
  - new ListUtils<T>(new ArrayList<T>) is verbose without any added benefit
  - If your module offers convenience methods for multiple tasks, you have serious cohesion problems you need to solve first
- Convenience methods extend a limited set of functionality
  - Client can easily implement it themselves, by why make them?
  - If you can’t control the parent class, you can use extension methods or the decorator pattern (as in the ListUtils example above)
- Template methods plug holes in algorithms
  - And they’re usually protected
  - The base class exposes the main API for clients
Other kinds of duplications

Duplication applies to more than just code

- **Logic.** How do we serialize and de-serialize our objects?
- **Configuration.** Database urls, passwords, resource locations
- **Policy.** For example, server timeouts, number of retries
- **Dependencies.** Which library are we using? Which abstractions are re-using?
- **Data format.** How do we canonize our data? How do we avoid *split-brain syndrome*?
Costs of code reuse

Code reuse is never free

1. Induces coupling
2. Increases complexity
3. Code duplication is cheaper than the wrong abstraction
Code reuse always induces coupling

- Either to the superclass (inheritance) or the delegated one (composition)
- But, as you might already expect, also increases cohesion and testability
Reusing code **always increases code complexity**
- More classes, more methods, more dependencies
  - Dependencies need to be injected, mocked, maintained
- Loss of code locality
- But also, increased abstraction *can* increase readability
“Code duplication is far cheaper than the wrong abstraction” (Sandi Metz)

- Avoid premature abstractions
  - Apply the Rule of Three
- Know the difference between accidental and inherent duplication
  - Do the code pieces only appear similar, or they do stem from the same source of truth?
  - Do the pieces have to change together?
Write code that is easy to delete, not easy to extend

- Start by writing ugly, copy-pasted, boiler-plate heavy code
  - This is sometimes called “exploratory code mode”
  - “A lot of programming is exploratory, and it’s quicker to get it wrong a few times and iterate than think to get it right first time”
- Refactor your code slowly, removing all the ugly parts, as you get to understand the design space better
  - Writing throw-away code is fine, committing to it isn’t
  - “Big balls of mud are the easiest to build but the most expensive to maintain”
- Focus on minimizing coupling and hiding details
  - “Although the single responsibility principle suggests that ‘each module should only handle one hard problem’, it is more important that ‘each hard problem is only handled by one module’”
- “if we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent"” (Dijkstra)
“Magic” literals

“Magic” or “naked” literals are literals which appear as input to methods and aren’t saved as interim variables

- Often considered an anti-pattern
  - As you were taught in Intro to CS
- But **inlining** variables can help increase readability and code locality; compare:

```java
server.setTimeoutInMillis(TIMEOUT_IN_MILLIS);
server.setTimeoutInMillis(100);
```

- If your variable name just repeats the literal value, what does it benefit?

```java
// Pick a better name or inline
private static final String TIMEOUT = "timeout";
// In case the value of 5 changes
private static final int FIVE = 5;
```
“Magic” literals (cont.)

- Extract literals to constants only to:
  1. Avoid `actual` duplication
  2. Give a value `important semantic` meaning

  - But it’s better if it’s obvious from context:

```javascript
// Bad
server.setTimeout(TIMEOUT);

// Okay
server.setTimeout(ONE_SECOND);

// Better
server.setTimeoutInMillis(1000);
```
“Magic” literals pretty

```
transactionsOnDate =
    populate("transactions/{sourceType}/on/{year}/{month}",
        params);
createTransaction = populate(
    "transactions/{sourceType}/{sourceId}/{targetType}/" +
    "{targetId}/{year}/{month}/" +
    "{percentage}",
        params);
deleteTransaction = populate(
    "transactions/{sourceType}/{sourceId}/{targetType}/" +
    "{targetId}/{year}/{month}",
        params);
```
“Magic” literals ugly

```java
private static final String TARGET = "target";
private static final String SOURCE = "source";
private static final String TYPE = "Type";
private static final String ID = "Id";
private static final String SEP = "/";
private static final String TRANSACTIONS =
    "transactions" + SEP;
private static final String TARGET_TYPE_AND_ID =
    SEP + var(TARGET + TYPE) + SEP + var(TARGET + ID) + SEP;
private static final String SOURCE_ID = var(SOURCE + ID);
private static final String SOURCE_TYPE = var(SOURCE + TYPE);
private static final String SOURCE_TYPE_AND_ID =
    SOURCE_TYPE + SEP + SOURCE_ID;
private static final String WHEN =
    var("year") + SEP + var("month");
private static final String SOURCE_TARGET_WHEN =
    SOURCE_TYPE_AND_ID + TARGET_TYPE_AND_ID + WHEN;
```
“Magic” literals ugly (cont.)

transactionsOnDate = populate(
    TRANSACTIONS + SOURCE_TYPE + SEP + "on" + SEP + WHEN,
    params);
createTransaction = populate(
    TRANSACTIONS + SOURCE_TARGET_WHEN + SEP + var("percentage"),
    params);
deleteTransaction = populate(
    TRANSACTIONS + SOURCE_TARGET_WHEN,
    params);

Which version do you think is better?