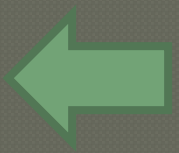
Design Patterns

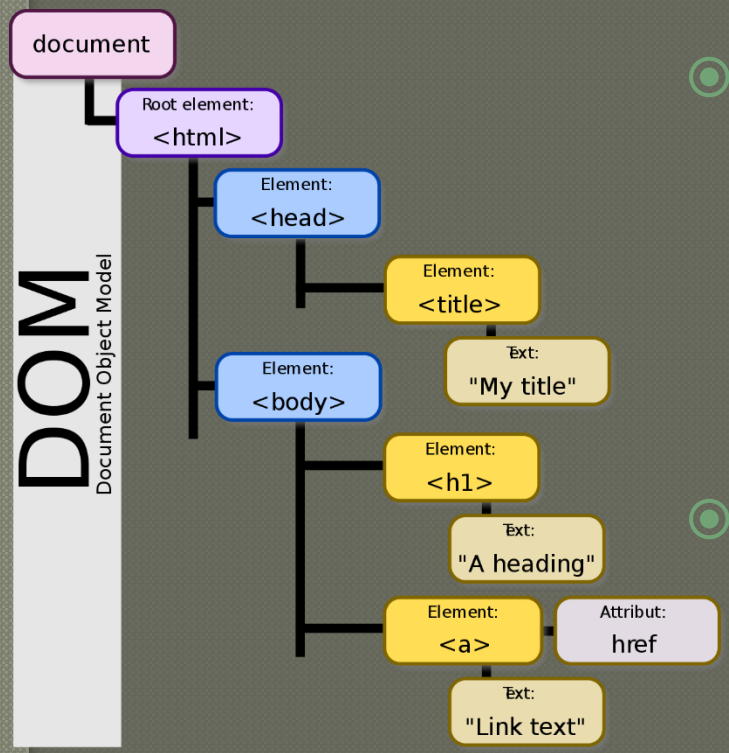
- The Strategy Pattern
- The Factory Method
- Generics
- The Abstract Factory Pattern
- The State Pattern
- The Observer Pattern
- The Adapter Pattern
- The Composite Pattern
- The Iterator Pattern
- **The Builder Pattern**
- Fallen Patterns
  - The Singleton Pattern
  - The Visitor Pattern



Builder

- A creational pattern
- Provides a convenient way to construct objects from configurations
- Provides parameter specific error checking in a convenient format

A Motivating Problem



- Suppose we have an HTML generator.
- We want to represent a tag as follows:
  - The tag name [required]
  - The class [optional]
  - The id [optional]
  - The style [optional]
- Example:

```
<div class = “text-region” id = “title-region” style = “color: red;”>…
```

A Motivating Problem

- Code?

```
class Tag {
    final String name;
    final String htmlClass;
    final String id;
    final String style;
    final Tag[] children;
}
```

- Constructors?
  - One for each combination of optional components

What are the Issues?

- We've made fields final, because changing them can cause an existing, correct DOM, to become invalid
- Allowing setters to change the fields requires cascading data integrity checks
- This causes a design problem:
  - We need to set the fields in the constructors
  - We either have a large, hard to remember constructor
  - Or we have 7 combinations of smaller constructors

Naïve Solution

- Let's just have a temporary Tag class
  - We intend to fill in the components we want into an instance, then pass the instance to the system.

```
class TagDesc {
    public string name;
    public string id;
    //…
}
```

- Usage:

```
TagDesc desc = new TagDesc();
desc.name = "div";
desc.id = "title-region";
//…
parentTag.addChild( desc );
```

What's Wrong with Naïve Solution?

Allows tags to be in an inconsistent state until they are submitted.

Error handling is more difficult and must be handled outside of the temporary class.

This isn't that bad, but we can make it better.

Using a Builder

- We build a tag in pieces:

```
Tag tag = Tag.builder("div")
          .htmlClass("text-region")
          .id("title-region")
          .style("color: red;")
          .build();
```

How Does It Work?

- The builder can be an inner class of Tag
  - useful if member variables are private
  - In our case, Builder does not need to be inner
- "builder()" is a method that returns a builder
- The Builder class contains methods for each optional element
  - Can also contain methods for required elements
  - It's up to you if you want required elements to be conventional parameters or not

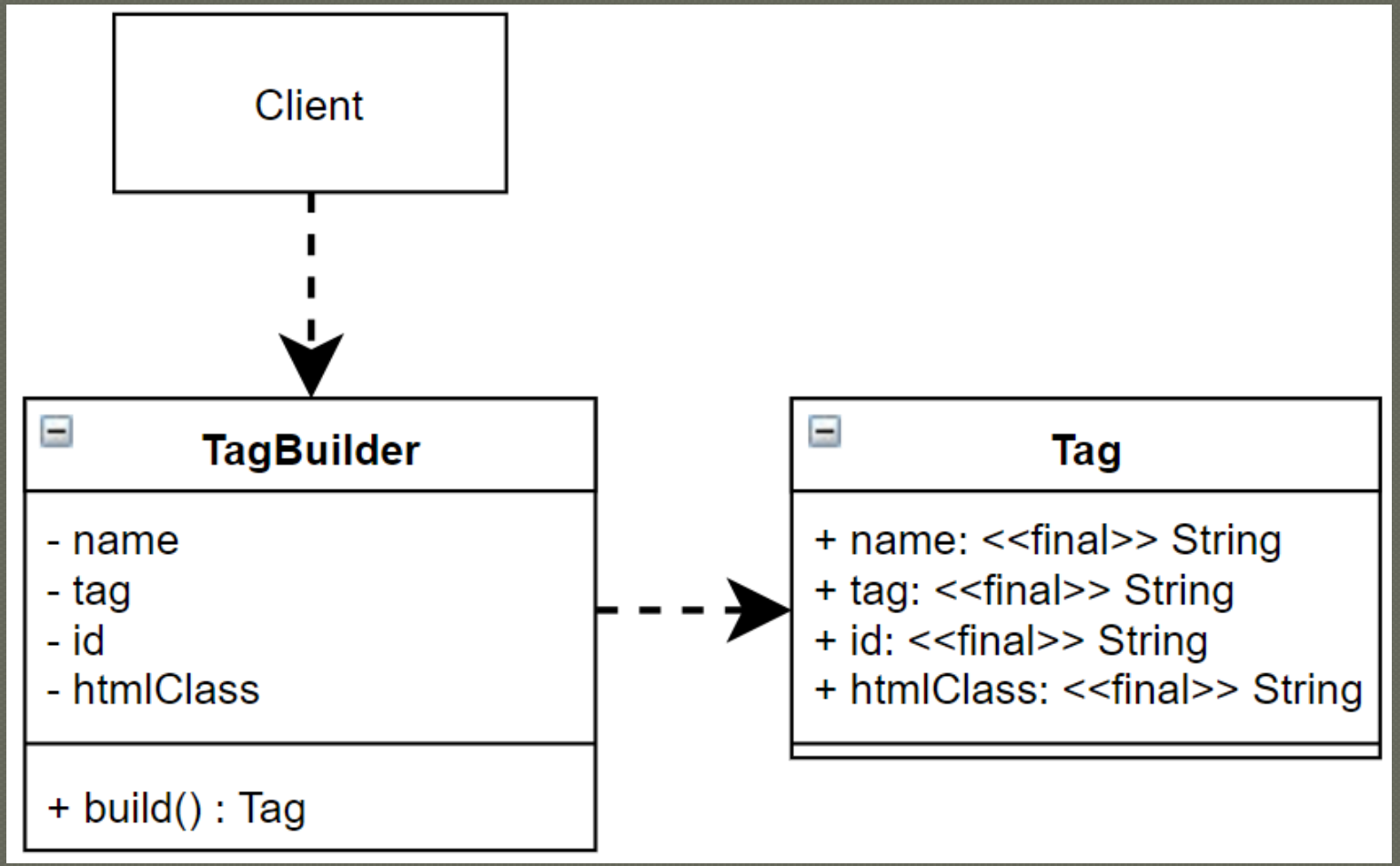
How Does It Work?

- The "build()" method returns the product (the actual Tag)
- The argument methods return the builder itself
- This is what allows the method chaining
- The fact that it's the same object means we're not really violating the Law of Demeter
- The actual fields are private

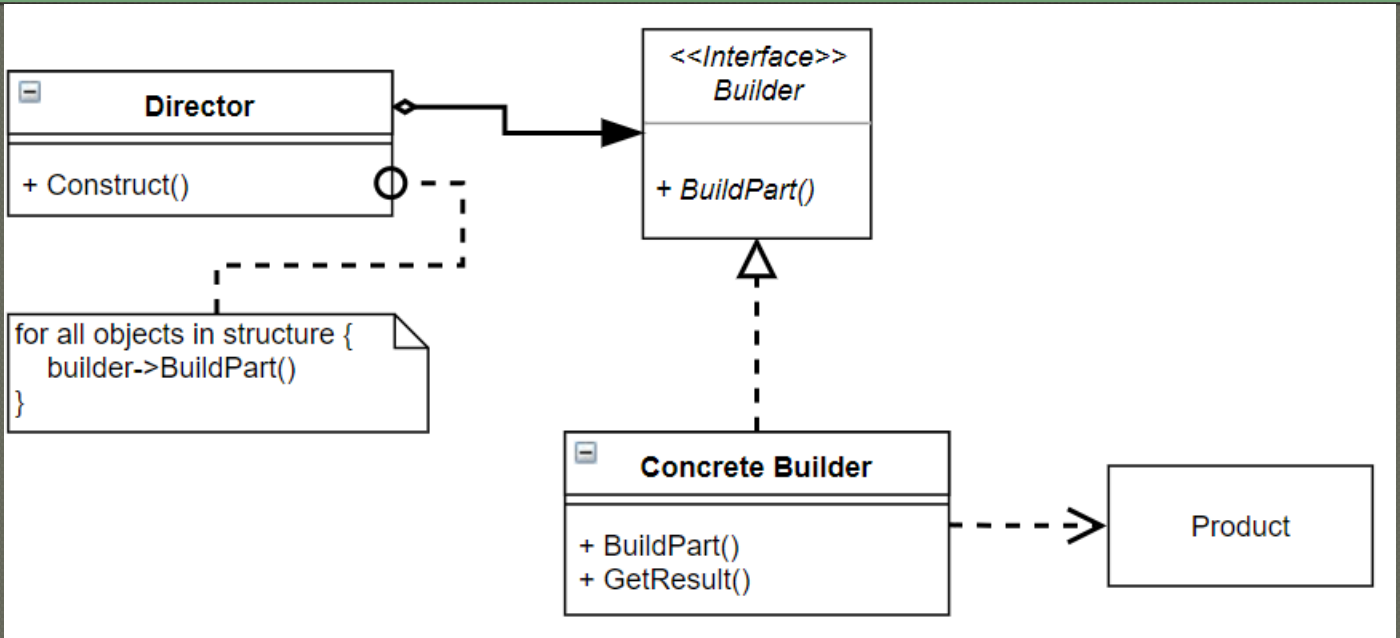
How Do We Benefit?

- Nice syntax
- Error checking is line-by-line

UML


Client
TagBuilder
- name
- tag
- id
- htmlClass
+ build() : Tag
Tag
+ name: <<final>> String
+ tag: <<final>> String
+ id: <<final>> String
+ htmlClass: <<final>> String

Extended Pattern



- Director is the "user" of the builder
- The concrete builder itself constructs the final object
- Inheritance is rarely used in practice with this pattern

Does it pass the YAGNI test?

- For well-used libraries and frameworks: very convenient
- For you class project: probably not worth it
  - The benefits over a simple description class are so minor they may not be worth the overhead of developing all the extra methods

Other Examples

- DL4J:
  - https://deeplearning4j.org/quickstart
- Rust SDL2:
  - https://github.com/Rust-SDL2/rust-sdl2/blob/master/examples/gfx-demo.rs
- Keras:
  - https://keras.io/
- Toy example:
  - https://jlordiales.me/2012/12/13/the-builder-pattern-in-practice/

Your Job

- See if you can apply builder in your project
- Compare it to the simple approach of an ordinary constructor
- Is it worthwhile to use?

- You have all of the patterns that you need to consider for your project implementation
- It's time to choose the two that works best with your design, fold them into the design, and implement those component