INFINUM

Software Design Patterns (in JS)

LUKA SKUKAN

01 INTRODUCTION

WHAT IS A PATTERN?

- Reusable solution to a class of problems
- Template to apply to problem
- There are also antipatterns

THE GOOD PARTS

- Proven to work
- Highly reusable and generic
- Offer a common vocabulary
- Clean, recognisable interface

THE BAD PARTS

- Misuse
- Overengineering
- Possibly not needed in your paradigm of choice

RULES OF THUMB

- Don’t start with a pattern, let it emerge (don’t solve with a pattern, let it be solved by one)
- Being able to apply it doesn’t mean that you should
- Don’t (intentionally) use a pattern you don’t understand

PATTERNS IN EVERYDAY JS

- Not a class-based language, but some patterns still apply
- Some popular libs (e.g. jQuery) use quite a few
- Some looks different than their counterparts in other languages

02

OO PATTERNS

CLASSIFICATION

- **Creational –** Creating an object
- **Structural –** Wraps, modifies or proxies an object
- **Behavioural –** Interaction between objects or with object
- **Concurrency** – Yeah, no

CREATIONAL: SINGLETON

- Ensures only one instance of object is constructed
- Typically via a getter method that does lazy initialisation once

```
const EventBusProvider = (function() {
 class EventBus {
  ...
 }

 let eventBus;

 return {
  instance() {
   if (!eventBus) {
    eventBus = new EventBus();
   }

   return eventBus;
  }
 };
}());
```

```
class KeypressHandler {
  constructor() {
    this.eventBus = EventBusProvider.instance();
  }
}
```

CREATIONAL: SINGLETON

- Ensures only one instance of object is constructed
- Typically via a getter method that does lazy initialisation once

SINGLETON

+ Good for shared, single-point-of-access objects, but...

- Basically a global object
- Why should an object know it's a singleton? Isn't that the application's job?
- Shared global state – testability, side effects...

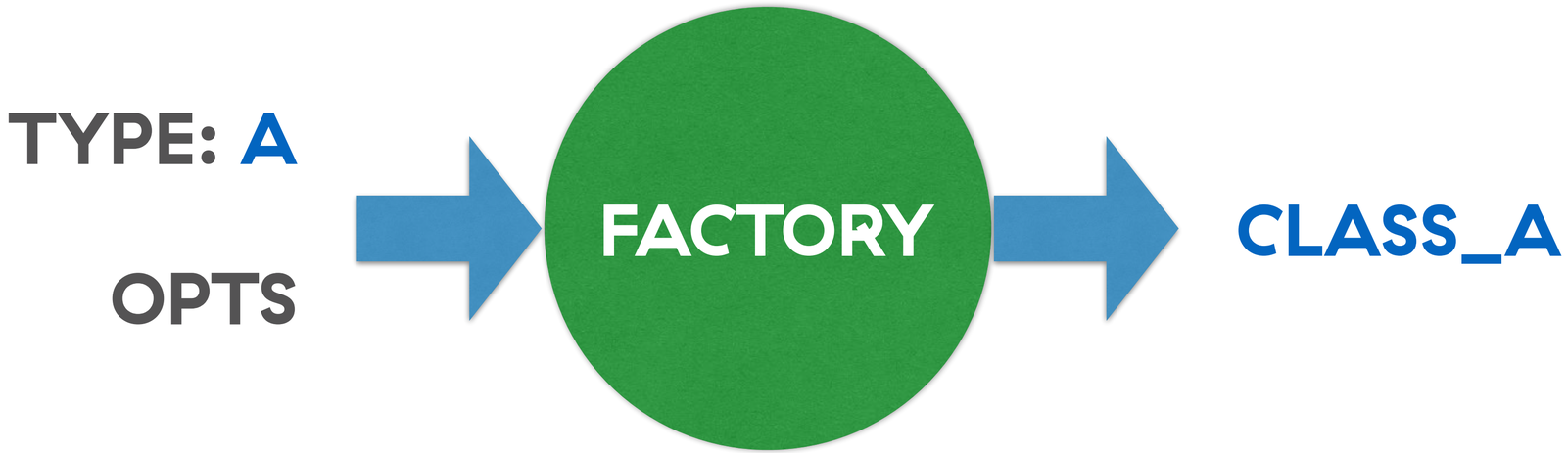
DEPENDENCY INJECTION

- Externalises the job of managing object instantiation
- Could be done via dependency system (e.g. RequireJS), string matching (Angular), ...

CREATIONAL: FACTORY

- Abstracts object creation
- Uniform interface for construction of similar objects (typically duck-typed in JS)

```
const formOpts = {
 style: ‘iOS’,
 color: ‘lightgrey’,
 model: someModel
};

const form = FormComponentFactory.create(
 ‘form’,
 ‘formName’,
 formOpts
);
const usernameInput = FormComponentFactory.create(
 ‘input’,
 ‘username’,
  formOpts
);
const submit = FormComponentFactory.create(
  ‘submit’,
  formOpts
);

form.attach([usernameInput, submit]);
```


TYPE: A
OPTS
FACTORY
CLASS_A


TYPE: B
OPTS
FACTORY
CLASS_B

FACTORY

+ Abstracts worrying about environment or complex construction logic
+ Can sometimes decouple logic

- However, often introduces overhead, makes testing more difficult

~ Usually an overkill for mere apps

CREATIONAL: BUILDER

- Abstracts object creation
- Start with a basic instance, add properties (typically through chaining)
- Popular in UI frameworks, jQuery

```
var prompt =
 makePrompt('Are you certain you want to do this?')
 .fullscreen()
 .confirmText('Sure')
 .cancelText('Nope!')
 .withCloseButton(true)
 .withOverlay(false)
 .onConfirm(confirm)
 .onCancel(cancel);
```

BUILDER

+ Good replacement for “telescoping constructor”
+ Great for widgets, unit tests, ...
+ Quite readable

- Many variants mean many code paths to test

~ Typically not app-level stuff

STRUCTURAL: COMPOSITE

- Same interface working with single instance or with collection
- Well-known example is jQuery
- Doesn’t have to be an array

```
// Same interface
$('#single').addClass('is-hidden');
$('.many').addClass('is-hidden');
```

COMPOSITE

+ Great for usability

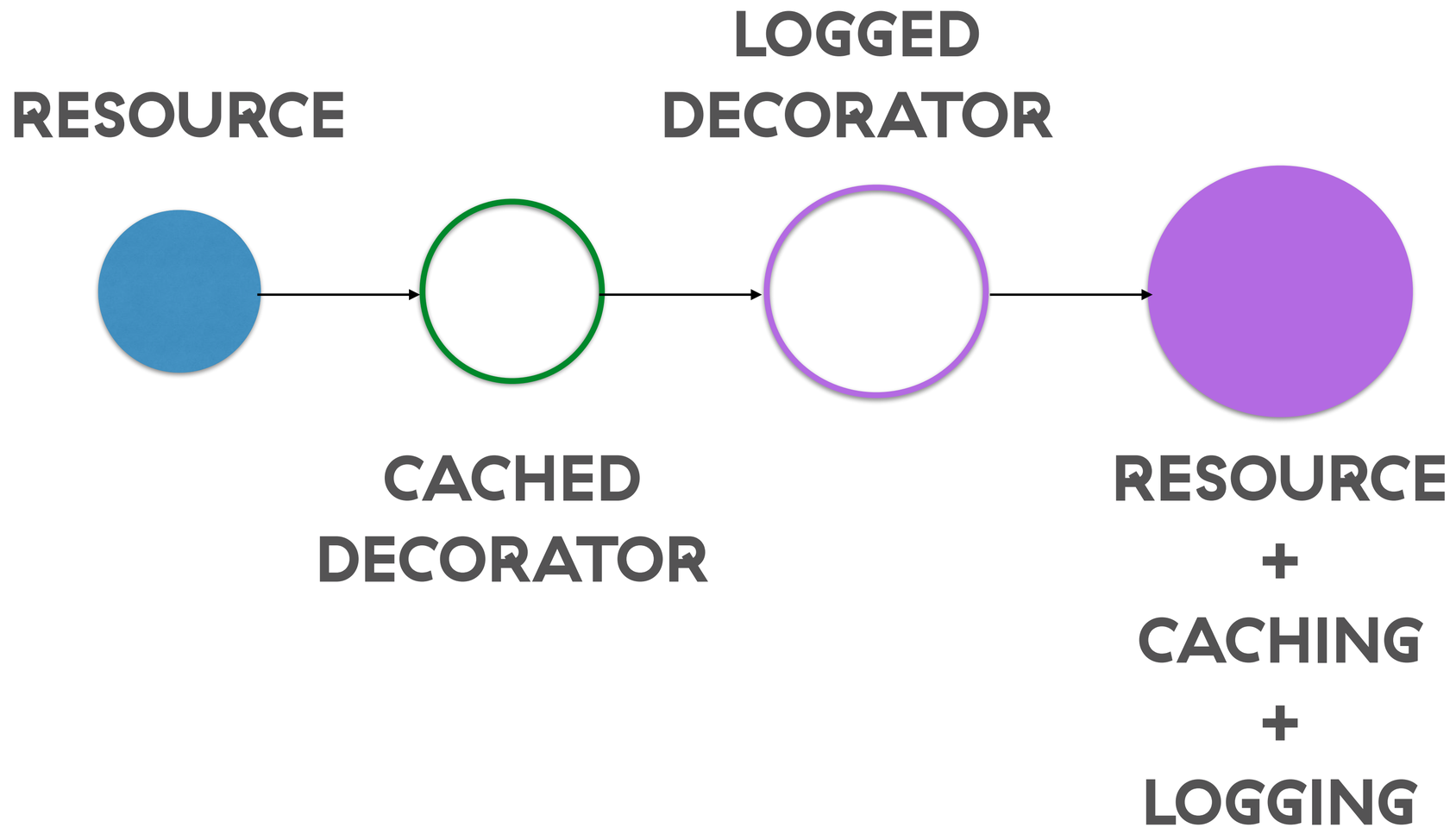
- Kinda sorta maybe obfuscates code a tiny bit, I guess?

STRUCTURAL: DECORATOR

- Dynamically extend functionality of already initialised object
- Are always given the object, never construct it
- Alternative to subclassing for extending functionality
- In JS, typically duck typed

```
class Resource {
 constructor(url) {...}
 all() {...}
 load(id) {...}
 save() {...}
 delete() {...}
}
```

```
class LoggedResource {
 constructor(resource) {
  this.resource = resource;
 }

 ...
 load(id) {
  logger.info('Loading', id);

  return this.resource.load(id);
 }
 ...
}
```


RESOURCE
LOGGED DECORATOR
CACHED DECORATOR
RESOURCE + CACHING + LOGGING

DECORATOR

+ Flexible
+ Can be nested indefinitely, like turtles
+ Avoids subclassing

- Harder to keep in your head, less efficient
- Potential scope cluttering with similar objects

STRUCTURAL: FASADE

- Simple interface for more complex logic
- Does stuff behind the curtains (housework, environment-specific, platform-specific...)
- Very big in jQuery, for example

```
// All facades
$('.xyz').on(...);
$('.xyz').css(...);
$.ajax(...);
```

FASADE

+ Much nicer interface, hide the mess inside

- Comes at a performance hit

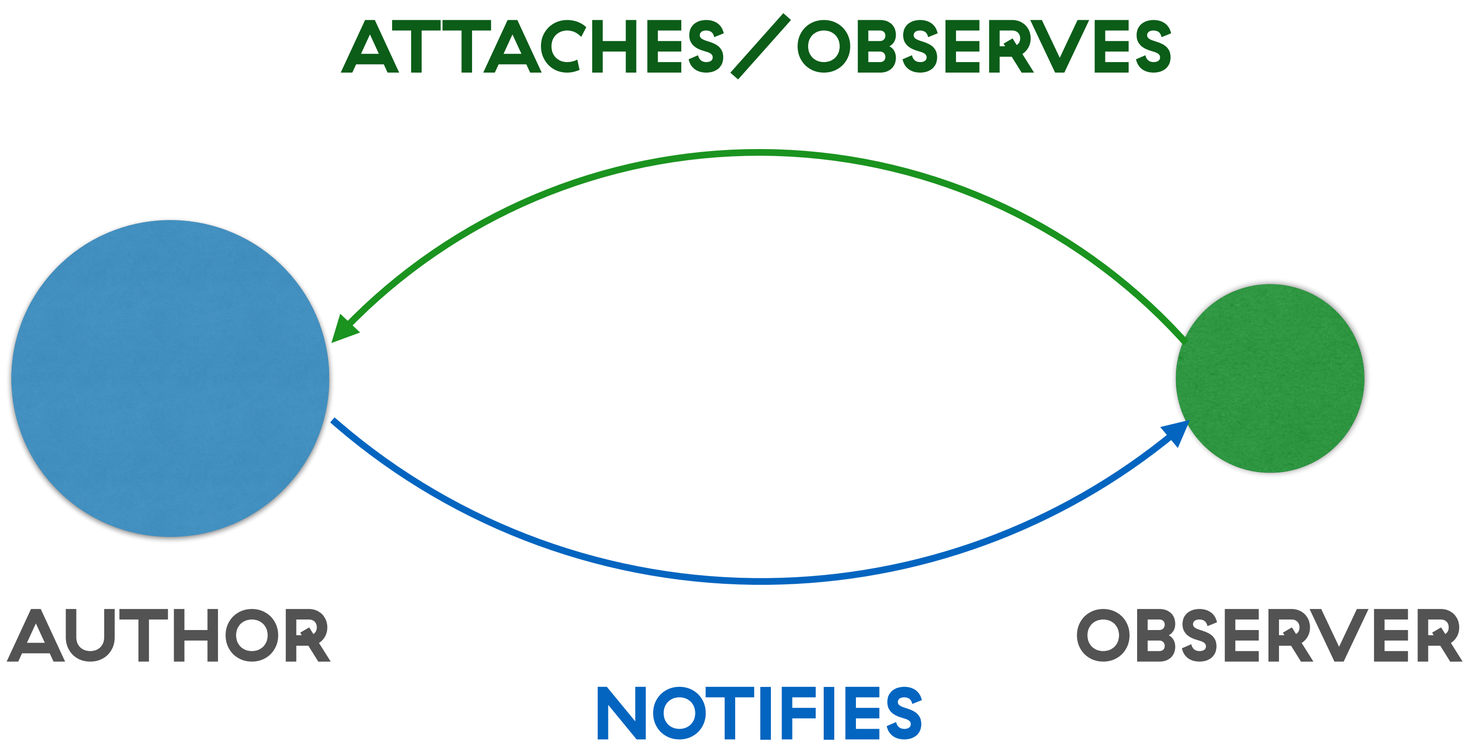
~ Uncommon in client code (what are you doing with your code?)

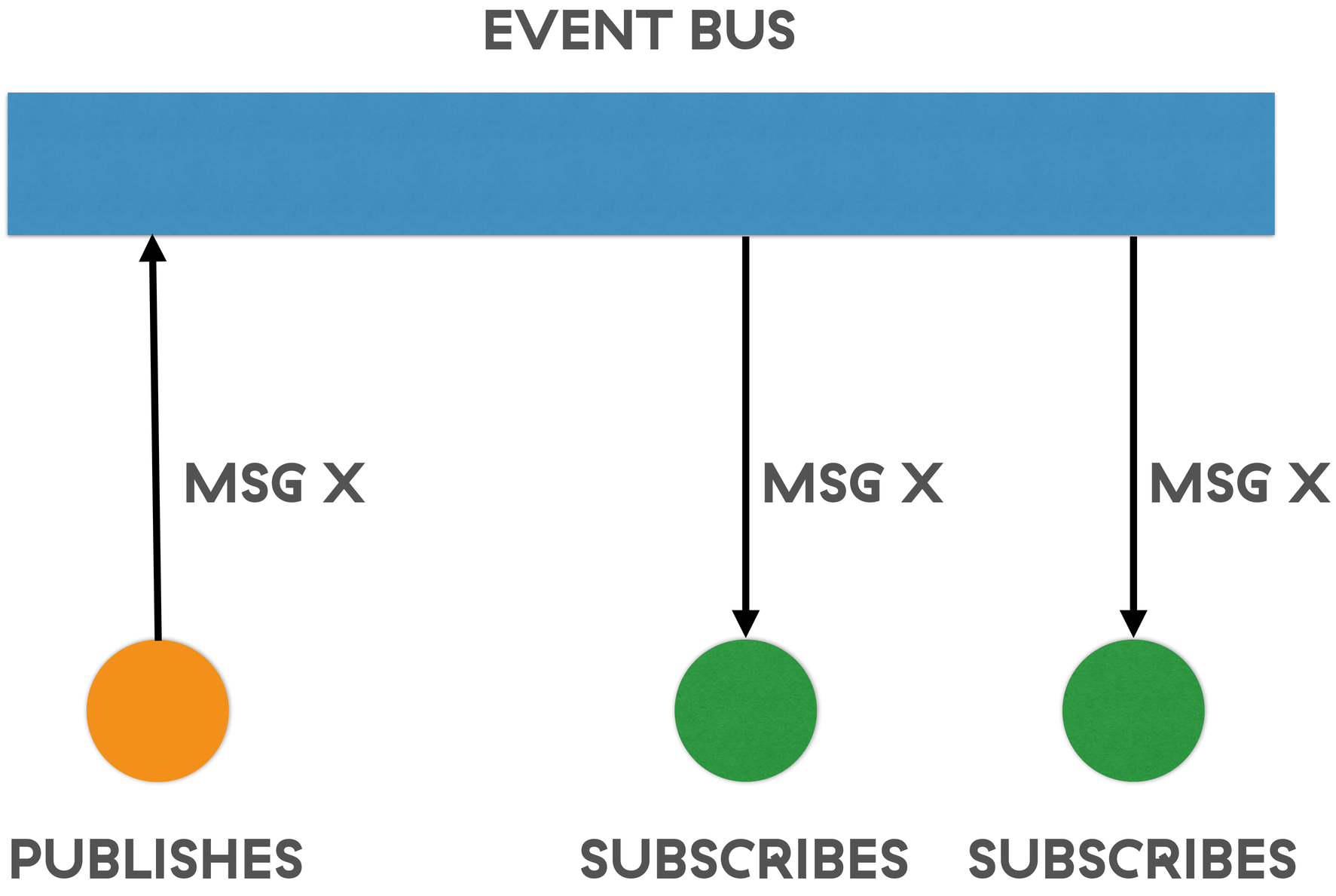
BEHAVIOURAL: OBSERVER + PUB/SUB

- Observers that need to know about event register with author
- Author has the responsibility of notifying them
- Observers know how to react
- Duck typing all around
- Pub/Sub is similar, but uses shared channel instead of explicit refs

```
class ExampleAuthor {
 constructor() {
  this.observers = [];
 }

 addObserver(o) {
  this.observers.push(o);
 }

 removeObserver(o) {
  ...
 }

 notifyObservers(e) {
  this.observers.forEach((o) => o.notify(e));
 }
 ...
}
```

```
class Observer {
 ...
 notify(event) {
  ...
 }
 ...
}
```


ATTACHES / OBSERVES
AUTHOR
OBSERVER
NOTIFIES


EVENT BUS
MSG X
MSG X
MSG X
PUBLISHES
SUBSCRIBES
SUBSCRIBES

BEHAVIOURAL: OBSERVER + PUB/SUB

- Observers that need to know about event register with author
- Author has the responsibility of notifying them
- Observers know how to react
- Duck typing all around
- Pub/Sub is similar, but uses shared channel instead of explicit refs

OBSERVER + PUB/SUB

+ Decouples semi-independent parts
+ Needs to know less about objects
+ Pub/sub especially decoupled

- Observers unaware of one another
- No guarantee of presence (testing unreliable)

BEHAVIOURAL: MIXIN

- Contains a bundle of common functionality
- Used to extend other objects
- Alternative to subclassing or, sometimes, utility classes
- Built-in in various languages as Mixins or Traits

```
class Rectangle extends Drawable {
  ...
}

class Path extends Drawable {
 ...
}

const AnimatedTrait = {
 fadeIn(duration) {
  ...
 }

 fadeOut(duration) {
  ...
 }

 moveTo(x, y, duration) {
  ...
 }
}
```

```
let rect = new Rectangle(...);
let path = new Path(...);
_.extend(rect, AnimatedTrait);
_.extend(path, AnimatedTrait);

rect.fadeIn(200);
path.fadeIn(300);
rect.moveTo(100, 100, 500);
```

BEHAVIOURAL: MIXIN

+ Good against code duplication
+ Good for modularisation
+ Testable (typically) and mockable

- Prototype pollution (if applied to classes instead of instances)

03 FINAL WORDS

DON’T LIMIT YOURSELF

- Fine and dandy, but not everything deserves a pattern application
- JS is a multiparadigm language, use functional idioms!
- Don’t make everything an object, this is not Java

KNOW YOUR PATTERNS

- Patterns are solutions to common problems. Knowing them still helps.
- Repeat: Great for communicating problems and solutions, cornerstone of software engineering
- Know when not to use them

LINKS

- https://addyosmani.com/resources/essentialjsdesignpatterns/book/ (Resource)
- http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf (Resource)
- http://www.norvig.com/design-patterns/design-patterns.pdf (Criticism)

INFINUM

Any questions?

Visit infinum.co or find us on social networks:

infinum.co infinumco infinumco infinum