# Patrones aplicados

Usé una combinación de los patrones strategy y factory para lograr el manejo de distintos buscadores sin agregar complejidad al desarrollo tratando de distinguir cuál esta siendo usado en ese momento.

## Factory Pattern

```
class SearchEngineFactory {
  constructor(url) {
    if (url.includes('google')) {
      return new GoogleEngine();
    } else if (url.includes('bing')) {
      return new BingEngine();
    } else if (url.includes('duck')) {
      return new DuckEngine();
    }
    return null;
  }
}
```

## Strategy Pattern

```
class BackgroundExtension extends AbstractP2PExtensionBackground {
  peers = [];
  searchEngine = new ResultGetter();
  currentTab = null;
```

```
class ResultGetter {
  constructor() {
    this.resultList = [];
  }
  setCurrentEngine(engine) {
    this.currentEngine = new SearchEngineFactory(engine);
  }
```

La clase BackgroundExtension se compone con un searchEngine, instancia de ResultGetter que contiene la lógica propia para interpretar los resultados obtenidos dependiendo del motor de búsqueda. Este currentEngine es actualizado mediante el método setCurrentEngine, que mediante la url recibida guardará una nueva instancia de ya sea GoogleEngine, DuckEngine o

BingEngine. Las tres clases heredan de SearchEngine, y obtendrán la información recibida de la query, conociendo la “estrategia” necesaria para cada uno.

La instancia guardada en currentEngine es determinada con un patrón Factory, que retorna en su constructor la instancia que corresponde.

En caso de que quisiese agregar un nuevo motor de búsqueda, como Yahoo o Ask, lo único que habría que actualizar es:

1. Crear una clase que herede de SearchEngine.
2. Implementar los métodos propios de las clases que extienden SearchEngine. (No los llamo abstractos porque JS no maneja nativamente ese concepto).
3. Incluir este caso en la clase SearchEngineFactory.

```
class GoogleEngine extends SearchEngine {
  constructor() {
    super();
    this.config = {
      inputQuery: "[name=q]",
      divListClass: "div[class=g]",
      linkResultClass: "div#rso>div>div>div>a",
      menuSelector: "div#hdtb-msb",
    };
    this._name = "google";
    this.external = ["bing", "duck"];
  }

  getUrl(query) {
    return `http://google.com/search?q=${query.replace(" ", "%20")}`;
  }

  getExternalResults(searchQuery) {
    const bing = new BingEngine();
    const duck = new DuckEngine();
    return Promise.all([
      bing.getOwnResults(searchQuery),
      duck.getOwnResults(searchQuery),
    ]);
  }

  getInnerText(node) {
    return node.childNodes[1].innerText;
  }
}
```

Ejemplo de implementación de Clase extendiendo SearchEngine

## Extra: Patron Template

```
112
113     getExternalResults() {
114       console.log("Subclass should implement");
115     }
116   }
```

En la clase SearchEngine, el método getExternalResults está solo escribiendo un comentario para que quien desarrolle lo implemente en la subclase. Lo agrego como extra porque al ser Javascript, la implementación sería mas correcta usando clases o métodos abstractos. Otra dirección que podría haber tomado es arrojar un error cuando se llega a llamar este método porque la subclase no lo sobre escribió.