# DWA_07.4 Challenge 1 (SOLID Principles)

- Which were the three best abstractions, and why?
    1. DOM Elements object

```
/**
 * All DOM elements
 * @typedef {Object} domElements
 */
const domElements = {
    dataListItems : document.querySelector('[data-list-items]'),
    dataListButton : document.querySelector('[data-list-button]'),
    dataListClose : document.querySelector('[data-list-close]'),
    dataListActive : document.querySelector('[data-list-active]'),
    dataListMessage : document.querySelector('[data-list-message]'),
    dataListBlur : document.querySelector('[data-list-blur]'),
    dataListImage : document.querySelector('[data-list-image]'),
    dataListTitle : document.querySelector('[data-list-title]'),
    dataListSubtitle : document.querySelector('[data-list-subtitle]'),
    dataListDescription : document.querySelector('[data-list-description]'),
    dataSearchGenres : document.querySelector('[data-search-genres]'),
    dataSearchAuthors : document.querySelector('[data-search-authors]'),
    dataSettingsTheme : document.querySelector('[data-settings-theme]'),
    dataSearchCancel : document.querySelector('[data-search-cancel]'),
    dataSearchOverlay : document.querySelector('[data-search-overlay]'),
    dataSearchTitle : document.querySelector('[data-search-title]'),
    dataSearchForm : document.querySelector('[data-search-form]'),
    dataSettingsCancel : document.querySelector('[data-settings-cancel]'),
    dataSettingsOverlay : document.querySelector('[data-settings-overlay]'),
    dataSettingsForm : document.querySelector('[data-settings-form]'),
    dataHeaderSearch : document.querySelector('[data-header-search]'),
    dataHeaderSettings : document.querySelector('[data-header-settings]')
}
```

In this abstraction, all of the DOM elements were placed into an object. This allows for all of the same instances to be grouped together and for more controlled access in other areas of the code. It can be further abstracted by adding nested objects of the DOM elements with similar data-keys.

2. One function, one purpose

```
domElements.dataListButton.addEventListener('click', () => {
    const fragment = document.createDocumentFragment()

    for (const { author, id, image, title } of matches.slice(page * BOOKS_PER_PAGE, (page + 1) * BOOKS_PER_PAGE)) {
        const element = document.createElement('button');
                // @ts-ignore
                element.classList = 'preview';
        element.setAttribute('data-preview', id);

        element.innerHTML = `
            <img
                class="preview__image"
                src="${image}"
            />

            <div class="preview__info">
                <h3 class="preview__title">${title}</h3>
                <div class="preview__author">${authors[author]}</div>
            </div>
        `;

        fragment.appendChild(element);
    };

    // @ts-ignore
    domElements.dataListItems.appendChild(fragment);
    page += 1;
});
```

This section of the code was one of the better areas of abstraction as the function created was meant to serve one purpose only: to create the html for the snippet of a book, including the name of the author and book, and cover image.

3. One function, multiple callbacks

```
/**
 * Function to close overlay boxes
 * @returns {boolean}
 */
const openFalse = (item) => {
    return item.open = false;
};

// @ts-ignore
domElements.dataSearchCancel.addEventListener('click', () => {
    openFalse(domElements.dataSearchOverlay);
});
```

This function was created with the functionality to close overlay boxes when the ‘cancel’ button was clicked. Because multiple overlays had to be closed (like the ‘search’, ‘theme’ and ‘preview’ overlays), its callback could be placed in more than one area of the code.

- Which were the three worst abstractions, and why?
    1. Extraction from objects with additional functions

```
for (const { author, id, image, title } of matches.slice(page * BOOKS_PER_PAGE, (page + 1) * BOOKS_PER_PAGE)) {
    const element = document.createElement('button');
            // @ts-ignore
            element.classList = 'preview';
```

The for-of loops had multiple properties extracted from an iterable (matches). This type of abstraction is not necessarily bad, but because another method was placed on the iterable, the functionality became overcrowded. If another method was added to the .slice() function, the maintainability and complexity of the object would increase.

    2. HTML duplication

```
// @ts-ignore
domElements.dataListButton.innerText = `Show more (${books.length - BOOKS_PER_PAGE})`;
// @ts-ignore
domElements.dataListButton.disabled = (matches.length - (page * BOOKS_PER_PAGE)) > 0;

// @ts-ignore
domElements.dataListButton.innerHTML = `
    <span>Show more</span>
    <span class="list__remaining"> (${(matches.length - (page * BOOKS_PER_PAGE)) > 0 ? (matches.length - (page * BOOKS_PER_PAGE)) : 0})</span>
`
```

Here, the same element has html created for it in more than one instance. The same logic is being used to check the amount of books remaining, and generates the html to be displayed. This creates unnecessary code duplication.

    3. Multiple fragments

```
const genreHtml = document.createDocumentFragment()
const firstGenreElement = document.createElement('option')
```

In the code, for genres, authors and the previews there are multiple fragment elements created. All have unique identifiers, but serve the same purpose.

- How can the three worst abstractions be improved via SOLID principles?
    1. Extraction from objects with additional functions

```
for (const { author, id, image, title } of matches.slice(page * BOOKS_PER_PAGE, (page + 1) * BOOKS_PER_PAGE)) {
    const element = document.createElement('button');
            // @ts-ignore
            element.classList = 'preview';
```

This could be further abstracted by taking the iterable (matches), and creating a separate variable that selects the required amount of books and stores it. This makes use of the Open-Closed principle, in the sense that the contents of the variable is not modified, but extended so that just its name replaces the complexity.

2. HTML duplication

```
// @ts-ignore
domElements.dataListButton.innerText = `Show more (${books.length - BOOKS_PER_PAGE})`;
// @ts-ignore
domElements.dataListButton.disabled = (matches.length - (page * BOOKS_PER_PAGE)) > 0;

// @ts-ignore
domElements.dataListButton.innerHTML = `
    <span>Show more</span>
    <span class="list__remaining"> (${(matches.length - (page * BOOKS_PER_PAGE)) > 0 ? (matches.length - (page * BOOKS_PER_PAGE)) : 0})</span>
`
```

As a solution to the duplication, most of the functionality could be placed together since they serve the same purpose. The Single Responsibility Principle (SRP) is applied here, which states that a class, a module, or a function should have one responsibility, and making changes to that one function should not affect anything else.

3. Multiple fragments

```
const genreHtml = document.createDocumentFragment()
const firstGenreElement = document.createElement('option')
```

One function could be created in order to generate a fragment with default content, and callbacks can be added where needed with different inputs to give the required outputs. This is in reference, in a sense, to the Liskov Substitution Principle (LSP), which states that the properties of the parent can be extended to the child (in this case, where the main function is called in different areas), and even replaced, provided that the main functionality of the parent is not affected.