Here's the converted markdown version of the document:

React Shopping Application - Architectural Design Document

1. Application Overview

This is a React-based e-commerce application that allows users to browse products, add them to a cart, and complete a checkout process. The application uses TypeScript for type safety and Tailwind CSS for styling.

2. Architecture

The application follows a component-based architecture typical of React applications, with the addition of the Context API for state management. It also incorporates several design patterns to enhance modularity and maintainability.

2.1 Key Components

- App (Entry Point)
- ProductList
- ProductDetails
- Cart
- Checkout
- Category
- OrderConfirmation
- EditOrder
- Header

2.2 State Management

The application uses React's Context API for state management, with separate contexts for different concerns:

- ProductContext
- CartContext
- UserContext
- OrderContext

Each context has a corresponding data provider that manages the state and provides methods to update it.

2.3 Routing

React Router is used for navigation between different views of the application.

```
<Router>
  <div className="container mx-auto p-4">
```

```
      <Header />
      <div className="flex flex-col md:flex-row">
        <Routes>
          <Route path="/" element={<ProductList />} />
          <Route path="/product/:id" element={<ProductDetails />} />
          <Route path="/category/:category" element={<Category />} />
          <Route path="/checkout" element={<Checkout />} />
          <Route path="/order-confirmation" element={<OrderConfirmation />}
/>
          <Route path="/edit-order" element={<EditOrder />} />
        </Routes>
        <Cart />
      </div>
    </div>
  </Router>
```

2.4 Data Flow

The application follows a unidirectional data flow:

- State is stored in context providers
- Components consume state from contexts
- User interactions trigger methods provided by contexts to update state
- State updates cause components to re-render with new data

3. Design Patterns

3.1 Singleton Pattern

Used for cart and user management to ensure a single instance throughout the application.

```
import { CartItem } from '../types';

export class CartSingleton {
  private static instance: CartSingleton;
  private items: CartItem[] = [];

  private constructor() {}

  public static getInstance(): CartSingleton {
    if (!CartSingleton.instance) {
      CartSingleton.instance = new CartSingleton();
    }
    return CartSingleton.instance;
  }

  public addItem(item: CartItem): void {
    const existingItem = this.items.find(i => i.id === item.id);
    if (existingItem) {
      existingItem.quantity += 1;
    } else {
      this.items.push(item);
    }
  }

  public removeItem(id: number): void {
    this.items = this.items.filter(item => item.id !== id);
  }

  public getItems(): CartItem[] {
    return this.items;
  }

  public clearCart(): void {
    this.items = [];
  }

  public getItemCount(): number {
    return this.items.reduce((acc, item) => acc + item.quantity, 0);
  }
}
```

3.2 Factory Pattern

Used for creating product objects.

```
import { Product } from '../types';

export class ProductFactory {
  createProduct(id: number, name: string, price: number, description: string, images: string[], category: string): Product {
    return { id, name, price, description, images, category };
  }
}
```

3.3 Adapter Pattern

Used to convert Product objects to CartItem objects.

```
import { CartItem, Product } from '../types';

export class CartAdapter {
  static toCartItem(product: Product, quantity: number = 1): CartItem {
    return {
      ...product,
      quantity,
      image: product.images[0],
    };
  }
}
```

4. Key Files and Their Responsibilities

- `src/App.tsx`: Main component, sets up routing and context providers
- `src/components/*.tsx`: Individual UI components
- `src/context/*.tsx`: Context definitions and hooks for state management
- `src/context/*DataProvider.ts`: Data providers for each context
- `src/patterns/*.ts`: Implementation of design patterns
- `src/types.ts`: TypeScript type definitions

5. Data Model

The application uses the following main data types:

- Product
- CartItem
- User
- Order

These are defined in the `src/types.ts` file.

6. Styling

The application uses Tailwind CSS for styling, configured in `tailwind.config.js` and applied in individual component files.

7. Build and Development

The application uses Vite as the build tool and development server, configured in `vite.config.ts`.

8. Future Considerations

- Implement server-side rendering for improved SEO and initial load performance
- Add unit and integration tests
- Implement error boundaries for better error handling
- Consider using a more robust state management solution (e.g., Redux) if the application grows in complexity
- Implement lazy loading for improved performance with a larger product catalog

This design document provides a high-level overview of the application's architecture. It can be expanded with more detailed information about each component, data flow diagrams, and specific implementation details as needed.