Programming Principles

Pentingnya Clean Code

- **Readability (Keterbacaan):** Mudah dipahami dan diikuti.
- **Maintainability (Kemudahan Maintenance):** Sederhana untuk diperbarui dan dikelola.
- **Scalability (Skalabilitas):** Mampu berkembang dan berevolusi.

Component Design

```
import React from 'react'

function Comp({ data }) {
  return (
    <div>
      {data.map((d, i) => {
        if (d.visible) {
          return (
            <div key={i}>
              {d.title.length > 10 ? `${d.title.substring(0, 10)}...` : d.title} -{' '}
              {d.desc.length > 15 ? `${d.desc.substring(0, 15)}...` : d.desc}
            </div>
          )
        }
        return null
      })}
    </div>
  )
}
```

Component Design

```
import React from 'react'

function ItemList({ items }) {
  return (
    <div>
      {items.map(
        (item, index) =>
          item.isVisible && <Item key={index} title={item.title} description={item.description} />
      )}
    </div>
  )
}
```

Component Design

```
function Item({ title, description }) {
  const truncatedTitle = truncateString(title, 10)
  const truncatedDescription = truncateString(description, 15)

  return (
    <div>
      {truncatedTitle} - {truncatedDescription}
    </div>
  )
}

function truncateString(str, maxLength) {
  return str.length > maxLength ? `${str.substring(0, maxLength)}...` : str
}
```

Component Design

- **Konvensi Penamaan:** Dalam kode yang bersih, `ItemList` dan `Item` lebih deskriptif dibandingkan dengan `Comp` dan `d`. Ini membuat lebih mudah untuk memahami tujuan dari setiap komponen.
- **Dekomposisi Fungsi:** Memecah fungsionalitas ke dalam komponen yang lebih kecil (`Item` dan `truncateString`) meningkatkan keterbacaan dan kemampuan untuk digunakan kembali.
- **Logika yang Disederhanakan:** Kode bersih menghindari ekspresi inline yang kompleks, membuatnya lebih mudah dibaca dan dipelihara.
- **Eksplisit:** Versi kode bersih membuat maksud kode lebih eksplisit, seperti `isVisible` daripada `visible`, membuat lebih jelas apa yang diwakili oleh setiap variabel.

Component Design

Keterbacaan dalam kode sangat penting untuk pemeliharaan dan kolaborasi. Kode yang bersih memungkinkan pengembang lain (dan diri Anda di masa depan) untuk memahami, memperbarui, dan memperluas basis kode dengan lebih efisien.

State Management

```
import React, { useState } from 'react'

function UserProfile() {
  const [user, setUser] = useState({
    name: '',
    email: '',
    age: 0,
    isSubscribed: false,
  })

  const handleInputChange = (e) => {
    setUser({ ...user, [e.target.name]: e.target.value })
  }

  const handleCheckboxChange = () => {
    setUser({ ...user, isSubscribed: !user.isSubscribed })
  }

  return (
    <div>
      <input name="name" value={user.name} onChange={handleInputChange} />
      <input name="email" type="email" value={user.email} onChange={handleInputChange} />
      <input name="age" type="number" value={user.age} onChange={handleInputChange} />
      <input
        name="isSubscribed"
        type="checkbox"
        checked={user.isSubscribed}
        onChange={handleCheckboxChange}
      />
    </div>
  )
}
```

State Management

```
import React, { useState } from 'react'

function UserProfile() {
  const [name, setName] = useState('')
  const [email, setEmail] = useState('')
  const [age, setAge] = useState(0)
  const [isSubscribed, setIsSubscribed] = useState(false)

  const handleInputChange = (e) => {
    const { name, value } = e.target
    if (name === 'name') setName(value)
    else if (name === 'email') setEmail(value)
    else if (name === 'age') setAge(value)
  }

  return (
    <div>
      <input name="name" value={name} onChange={handleInputChange} />
      <input name="email" type="email" value={email} onChange={handleInputChange} />
      <input name="age" type="number" value={age} onChange={handleInputChange} />
      <input
        name="isSubscribed"
        type="checkbox"
        checked={isSubscribed}
        onChange={() => setIsSubscribed(!isSubscribed)}
      />
    </div>
  )
}
```

Pentingnya Manajemen State yang Lebih Baik

- **Pemisahan State:** Dalam kode yang bersih, variabel state dipisahkan (name, email, age, isSubscribed), tidak seperti kode yang tidak bersih yang menggabungkannya ke dalam satu objek state. Pemisahan ini mempermudah manajemen state dan membuat kode lebih mudah dibaca.
- **Logika Pembaruan yang Disederhanakan:** Kode yang bersih menggunakan fungsi setter individu untuk setiap variabel state, membuatnya lebih mudah untuk melacak dan mengelola perubahan state.

Pentingnya Manajemen State yang Lebih Baik

- **Modularitas:** Dengan mengelola setiap bagian state secara terpisah, kode menjadi lebih modular dan lebih mudah untuk dimodifikasi atau diperluas.
- **Optimasi Kinerja:** Memisahkan state juga dapat mengarah pada kinerja yang lebih baik, karena pembaruan pada variabel state individu tidak akan menyebabkan komponen yang bergantung pada state yang tidak terkait melakukan render ulang secara tidak perlu.

Praktik-praktik dalam manajemen state ini mengarah pada aplikasi React yang lebih mudah dipelihara dan diskalakan, terutama saat kompleksitas aplikasi bertambah.

Reusability

```
import React from 'react'

function UserList() {
  const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
  ]

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  )
}
```

Reusability

```
const List = ({ items, renderItem }) => {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{renderItem(item)}</li>
      ))}
    </ul>
  )
}

const User = List() => {
  const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
  ]

  return <List items={users} renderItem={(user) => user.name} />
}
```

Reusability

- **Keterpisahan Fungsionalitas:** Dalam kode yang reusable, komponen `List` terpisah dari data yang spesifik dan dapat digunakan dengan berbagai jenis data.
- **Fleksibilitas:** `List` menerima `items` dan `renderItem` sebagai prop, memungkinkan penggunaan yang fleksibel dalam berbagai situasi.

Reusability

- **Pengurangan Duplikasi:** Dengan memiliki komponen `List` yang reusable, kita menghindari duplikasi kode saat kita perlu menampilkan daftar item lain dalam aplikasi.
- **Pemeliharaan yang Lebih Mudah:** Dengan kode yang lebih modular dan reusable, pemeliharaan dan pembaruan pada aplikasi menjadi lebih sederhana dan terpusat.

Praktik ini menunjukkan bagaimana kode yang bersih dan terorganisir dapat meningkatkan kereusabilan dan efisiensi dalam pengembangan aplikasi.

Code Splitting and Lazy Loading

```
import React, { Suspense } from 'react'
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom'

const Home = React.lazy(() => import('./Home'))
const About = React.lazy(() => import('./About'))

function App() {
  return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
        </Switch>
      </Suspense>
    </Router>
  )
}
```

Effective Use of Hooks

```
import React, { useState, useEffect } from 'react'

function UserStatus({ userId }) {
  const [user, setUser] = useState(null)

  useEffect(() => {
    const fetchUser = async () => {
      const response = await fetch(`https://api.example.com/users/${userId}`)
      const userData = await response.json()
      setUser(userData)
    }

    fetchUser()
  }, [userId]) // Dependency array ensures effect runs when userId changes

  if (!user) {
    return <div>Loading user...</div>
  }

  return <div>Welcome, {user.name}!</div>
}
```

Directory Structure

```
ReactJS-App/
├── src/
│   ├── components/
│   │   ├── Common/
│   │   ├── Layout/
│   │   └── [SpecificFeature]/
│   ├── pages/
│   │   ├── HomePage/
│   │   ├── AboutPage/
│   │   └── [OtherPages]/
│   ├── hooks/
│   ├── context/
│   ├── services/
│   ├── utils/
│   ├── assets/
│   ├── styles/
│   ├── constants/
│   └── types/ (if using TypeScript)
├── public/
├── tests/
│   ├── components/
│   ├── hooks/
│   └── utils/
├── config/
├── docs/
├── .env (multiple .env files for different environments)
└── README.md
```

Directory Structure

Struktur direktori yang baik sangat penting dalam proyek React untuk memudahkan pengelolaan dan pemahaman kode. Berikut adalah beberapa prinsip dasar dalam mengorganisir file dan folder:

1. **Pemisahan Berdasarkan Fungsi:** Pisahkan kode berdasarkan fungsinya, misalnya, komponen, utilitas, hook, dan layanan. Ini mempermudah pencarian dan pemeliharaan kode.
2. **Modularitas:** Setiap komponen harus memiliki direktori sendiri jika memiliki file terkait, seperti stylesheet dan tes. Misalnya, direktori `Button` dapat berisi `Button.jsx`, `Button.test.jsx`, dan `Button.css`.

Directory Structure

3. **Direktori Umum:** Gunakan direktori umum untuk aset, utilitas, dan komponen yang digunakan secara luas di seluruh aplikasi.
4. **Pemisahan Halaman dan Komponen:** Pisahkan halaman dan komponen UI. Halaman adalah komponen yang terhubung langsung dengan rute, sementara komponen UI adalah bagian yang dapat digunakan kembali.
5. **Penamaan yang Konsisten:** Gunakan konvensi penamaan yang konsisten untuk file dan direktori untuk memudahkan pemahaman.

Component Organization

Organisasi komponen adalah tentang bagaimana mengelompokkan dan mengatur komponen dalam proyek React:

1. **Komponen Atomik vs Kompleks:** Pisahkan komponen kecil dan reusable (seperti tombol, input) dengan komponen yang lebih kompleks (seperti formulir, header).
2. **Komponen Kontainer vs Presentasional:** Komponen kontainer mengelola logika dan data, sementara komponen presentasional hanya bertanggung jawab atas tampilan UI.

Component Organization

3. **Pengelompokan Berdasarkan Fitur:** Kelompokkan komponen berdasarkan fitur atau fungsionalitas. Misalnya, semua komponen terkait dengan pengelolaan pengguna dapat ditempatkan dalam satu grup.
4. **Reusability:** Buat komponen sedemikian rupa sehingga mereka dapat digunakan kembali di tempat lain dalam aplikasi.

Mengelola struktur direktori dan organisasi komponen dengan baik dapat meningkatkan efisiensi dan kemudahan dalam pengembangan dan pemeliharaan aplikasi React.

KISS (Keep It Simple, Stupid)

Prinsip KISS menekankan pentingnya kesederhanaan dalam desain dan pengembangan. Intinya adalah:

- **Menghindari Kompleksitas yang Tidak Perlu:** Solusi yang lebih sederhana seringkali lebih baik. Hindari over-engineering atau menambahkan fitur yang tidak esensial.
- **Mudah Dipahami dan Dikelola:** Kode yang sederhana lebih mudah untuk dipahami, dikelola, dan dikembangkan.

DRY (Don't Repeat Yourself)

Prinsip DRY berkaitan dengan pengurangan duplikasi dalam kode:

- **Menghindari Duplikasi:** Setiap bagian informasi atau logika dalam kode harus ada hanya dalam satu tempat. Jika ada kode yang serupa atau berulang, itu harus diekstrak menjadi komponen, fungsi, atau modul yang reusable.
- **Meningkatkan Efisiensi:** Mengurangi duplikasi memudahkan dalam pemeliharaan dan mengurangi kesalahan, karena perubahan hanya perlu dilakukan di satu tempat.

YAGNI (You Aren't Gonna Need It)

YAGNI adalah prinsip yang menyarankan untuk tidak menambahkan fungsi sampai fungsi tersebut benar-benar diperlukan:

- **Hindari Over-Engineering:** Jangan membangun fitur atau fungsi berdasarkan antisipasi kebutuhan masa depan yang belum pasti.
- **Fokus pada Kebutuhan Saat Ini:** Konsentrasikan sumber daya pada apa yang benar-benar dibutuhkan saat ini, yang akan membantu menjaga proyek tetap ringkas dan efisien.

Penerapan ketiga prinsip ini - KISS, DRY, dan YAGNI - membantu dalam menciptakan kode yang lebih bersih, mudah dipelihara, dan efisien.

SOLID Principles

Memang ada perdebatan di kalangan pengembang tentang seberapa relevan prinsip SOLID dalam konteks React. SOLID adalah kumpulan prinsip yang dirancang untuk pengembangan perangkat lunak berorientasi objek, dan ini bisa jadi tidak langsung terlihat bagaimana mereka diterapkan dalam pengembangan komponen React, yang lebih dekat ke paradigma fungsional.

Namun, beberapa aspek dari prinsip SOLID bisa relevan dan bermanfaat bahkan dalam pengembangan React:

SOLID Principles

Single Responsibility Principle (SRP)

SRP menyatakan bahwa sebuah kelas harus memiliki alasan perubahan yang tunggal. Dalam konteks React, ini bisa diartikan sebagai setiap komponen harus memiliki satu tanggung jawab dan tidak melakukan lebih dari yang seharusnya. Ini serupa dengan konsep komponen yang terfokus dan reusable.

SOLID Principles

Open/Closed Principle (OCP)

Prinsip ini menyatakan bahwa entitas perangkat lunak harus terbuka untuk ekstensi, tetapi tertutup untuk modifikasi. Dalam React, ini bisa diinterpretasikan sebagai ide untuk membuat komponen yang dapat diperluas melalui props tanpa perlu mengubah komponen itu sendiri.

SOLID Principles

Liskov Substitution Principle (LSP)

LSP berkaitan dengan substitusi kelas turunan. Ini kurang relevan di React, karena React lebih fungsional daripada berorientasi objek. Namun, konsep yang serupa dapat diterapkan dalam konteks penggunaan komponen.

SOLID Principles

Interface Segregation Principle (ISP)

ISP menyatakan bahwa klien tidak harus tergantung pada antarmuka yang tidak mereka gunakan. Dalam React, ini bisa diartikan sebagai penggunaan props secara bijaksana dan menghindari mengirimkan props yang tidak diperlukan ke komponen.

SOLID Principles

Dependency Inversion Principle (DIP)

DIP menekankan bahwa modul tingkat tinggi tidak harus bergantung pada modul tingkat rendah; keduanya harus bergantung pada abstraksi. Meskipun ini adalah konsep berorientasi objek, ide serupa bisa diterapkan dalam pengelolaan state dan logika bisnis di React, mendorong keterpisahan antara UI dan logika bisnis.

SOLID Principles

Jadi, meskipun prinsip SOLID tidak sepenuhnya sesuai atau langsung diterapkan dalam React, beberapa aspek dari prinsip-prinsip tersebut dapat digunakan untuk membimbing desain dan arsitektur komponen React yang lebih bersih dan terorganisir. Pendekatan ini memerlukan interpretasi dan adaptasi dari prinsip tersebut untuk sesuai dengan paradigma dan pola React.

Domain-Driven Design (DDD)

Domain-Driven Design (DDD) adalah pendekatan dalam pengembangan perangkat lunak yang menekankan pentingnya pemahaman domain atau bidang masalah yang akan diselesaikan. Berikut adalah penjelasan singkat tentang DDD:

Domain-Driven Design (DDD)

Konsep Dasar

- **Fokus pada Domain:** DDD berfokus pada pemodelan domain bisnis dan prosesnya. Ini melibatkan kolaborasi erat antara pengembang perangkat lunak dan ahli domain untuk memastikan bahwa software mencerminkan realitas kompleksitas dunia nyata.
- **Ubiquitous Language:** Menggunakan bahasa yang sama antara pengembang dan pemangku kepentingan bisnis untuk mendeskripsikan semua aspek domain, mengurangi ambiguitas dan meningkatkan komunikasi.

Domain-Driven Design (DDD)

Elemen Utama

- **Entity dan Value Object:** Mengidentifikasi objek dalam domain, dengan membedakan antara entity (mempunyai identitas yang unik) dan value object (didefinisikan oleh atributnya, bukan identitas).
- **Aggregate:** Kumpulan objek yang dikelola bersama dan dianggap sebagai satu unit untuk tujuan data changes.
- **Repository:** Menyediakan cara untuk mengakses objek domain, biasanya entity atau aggregate.
- **Service:** Menangani operasi domain yang tidak secara alami berada dalam objek domain.

Domain-Driven Design (DDD)

Tujuan

- **Solusi yang Sesuai dengan Kebutuhan Bisnis:** DDD bertujuan untuk menghasilkan desain yang lebih intuitif dan berorientasi pada kebutuhan bisnis, sehingga menghasilkan software yang lebih efektif dalam mendukung proses bisnis.
- **Keluwesan dan Skalabilitas:** Mempermudah evolusi dan adaptasi sistem karena perubahan dalam bisnis dan teknologi.

Domain-Driven Design (DDD)

Implementasi

- **Iteratif dan Kolaboratif:** Proses pengembangan DDD adalah iteratif dan melibatkan kolaborasi yang erat antara tim pengembang dan pemangku kepentingan bisnis.
- **Pemisahan Konteks:** DDD sering menerapkan konsep bounded context, yang membatasi model domain ke area spesifik fungsi bisnis, memudahkan pemahaman dan pengelolaan.

Domain-Driven Design adalah pendekatan yang kuat untuk mengembangkan sistem kompleks, di mana fokus pada domain bisnis dan kolaborasi antara tim teknis dan non-teknis sangat penting untuk kesuksesan proyek.

Domain-Driven Design (DDD)

Example: E-Commerce Application

Misalkan kita sedang mengembangkan aplikasi e-commerce. Dalam konteks DDD, kita akan memulai dengan memahami domain bisnis dan menetapkan Ubiquitous Language.

Domain-Driven Design (DDD)

Step 1: Defining the Domain Model

- **Entities:** `Product`, `User`, `Order`.
- **Value Objects:** `Address`, `Price`.
- **Aggregates:** Shopping Cart (combining `Products`, `Quantity`, and `Price`).
- **Repositories:** For accessing and storing entity data like `ProductRepository`, `UserRepository`.
- **Services:** Services such as `AuthenticationService`, `OrderService`.

Domain-Driven Design (DDD)

Step 2: Breaking Down into React Components

- **Product Component:** Displaying product information.
- **Shopping Cart Component:** Handling shopping cart logic.
- **Checkout Component:** Processing payments and order confirmation.

Domain-Driven Design (DDD)

Step 3: Directory Structure Based on Domain

```
src/
|-- products/
|   |-- ProductList.jsx
|   |-- ProductDetail.jsx
|   |-- ProductRepository.js
|-- cart/
|   |-- ShoppingCart.jsx
|   |-- CartItem.jsx
|-- users/
|   |-- UserProfile.jsx
|   |-- AuthenticationService.js
|-- orders/
    |-- Checkout.jsx
    |-- OrderService.js
```

Domain-Driven Design (DDD)

Step 4: Applying Bounded Context

- **Separating Business Logic:** Each directory like `products`, `cart`, `users`, and `orders` reflects a different bounded context.
- **Modularity:** Each module focuses on one aspect of the domain and is isolated from others, making it easier to manage and develop.

> Bounded context adalah konsep dalam DDD yang membatasi model domain ke area spesifik fungsi bisnis, memudahkan pemahaman dan pengelolaan.

Domain-Driven Design (DDD)

Step 5: Communication Between Components

- **Props and State:** Use props and state to manage data and interactions between components.
- **Context API or State Management Library:** For more complex cross-component or context state management, like Redux or the Context API.

Dalam penerapan ini, fokus utama adalah mengorganisir kode React sedemikian rupa sehingga mencerminkan struktur domain bisnis. Ini memudahkan pengembang untuk memahami bagaimana aplikasi bekerja dari sudut pandang bisnis dan membuat kode lebih terorganisir dan mudah dipelihara.