Fragen zum Thema Code "Smells"

Von Klemens Gassner

„Bei der Herstellung dieses Textes [oder wahlweise Bildes oder des Programmiercodes etc.] wurde ChatGPT eingesetzt. Mit folgenden Prompts habe ich die KI gesteuert: "1. Parallelität bei Programmen Was versteht man darunter? Anwendungsbereiche und Beispiele? Welche Vor- und Nachteile ergeben sich aus der Parallelität von Programmen., 2. Verteilte Systeme? Was versteht man darunter? Anwendungsbereiche und Beispiele? Welche Vor- und Nachteile ergeben sich aus der Verteilung von Programmen., 3. Was versteht man unter einem Prozess? Welche Eigenschaften hat ein Prozess? Was versteht man unter einem Thread? Zusammenhang/Unterscheidung von Prozess und Thread?, 4. Was versteht man unter Multithreading und was sind desen Vor und Nachteile?“1. Duplicate Code: Vermeiden Sie doppelten Code, indem Sie eine Funktion oder eine Klasse erstellen und den Code darin wiederverwenden. Eine andere Möglichkeit ist die Verwendung von Vererbung, um gemeinsame Funktionalitäten in einer Basis-Klasse zu definieren und diese in abgeleiteten Klassen zu erweitern.

1. Parallelität bei Programmen

1. Wie können die folgenden typischen "Fehler" beim Programmieren vermieden werden (Code Smells)
    - Long Methods: Aufteilen von langen Methoden in kleinere, wiederverwendbare Funktionen. Stellen Sie sicher, dass jede Funktion nur eine einzige Aufgabe hat und gut benannt ist.
    - Large Classes: Versuchen Sie, eine Klasse auf eine einzige Verantwortlichkeit zu beschränken und delegieren Sie andere Aufgaben an andere Klassen. Vermeiden Sie auch die Verwendung von globalen Variablen oder Funktionen, die sich auf viele verschiedene Klassen auswirken.
    - Shotgun surgery: Konsolidieren Sie Code, der in vielen verschiedenen Klassen oder Methoden dupliziert ist, in einer einzigen Stelle. Wenn Sie Änderungen an dieser Funktion vornehmen, müssen Sie nur an einer einzigen Stelle Änderungen vornehmen, anstatt an vielen verschiedenen Stellen.
    - Feature envy: Wenn eine Methode in einer Klasse mehr Zeit damit verbringt, auf die Daten einer anderen Klasse zuzugreifen, sollten Sie wahrscheinlich die Funktionalität in die entsprechende Klasse verschieben. Stellen Sie sicher, dass jede Klasse ihre eigenen Daten verarbeitet und so wenig wie möglich von anderen Klassen abhängt.
2. Was versteht man unter den folgenden Ausdrücken im Zusammenhang mit Code smells (Nennen Beispiele hierfür)
    - Object-Orientation Abusers: Diese Code Smells treten auf, wenn Entwickler grundlegende Prinzipien der objektorientierten Programmierung (OOP) missachten. Beispiele hierfür sind Klassen, die zu viele Verantwortlichkeiten haben (z.B. eine Klasse, die sowohl für die Verarbeitung von Daten als auch für die Darstellung von Benutzeroberflächen zuständig ist), oder die Verwendung von Vererbung, wo dies nicht angebracht ist. OOP-Abuser-Smells führen dazu, dass der Code schwer zu verstehen, zu testen und zu warten ist. Beispiel: Eine Klasse, die sowohl für die Verarbeitung von Daten als auch für die Erstellung von Benutzeroberflächen zuständig ist. In diesem Fall sollte die Klasse aufgeteilt werden, um nur eine Verantwortlichkeit zu haben.

- Change Preventers: Diese Code Smells treten auf, wenn der Code so geschrieben ist, dass er Änderungen verhindert oder erschwert. Beispiele hierfür sind hartcodierte Werte, die an vielen Stellen im Code verwendet werden, oder das Festlegen von Funktionalitäten in Beton gegossenen Strukturen, die schwer zu ändern sind. Änderungsverhindernde Code-Smells führen dazu, dass der Code schwer zu warten und zu aktualisieren ist. Beispiel: Ein hartcodierter Dateipfad, der an vielen Stellen im Code verwendet wird. Wenn der Pfad geändert werden muss, muss der Code an vielen verschiedenen Stellen aktualisiert werden.

- Dispensables: Diese Code Smells treten auf, wenn der Code unnötig ist oder nicht ausgeführt wird. Beispiele hierfür sind tote oder unbenutzte Codeabschnitte, redundante Funktionen oder ungenutzte Variablen. Entfernung von unnötigem Code kann die Lesbarkeit und Wartbarkeit verbessern. Beispiel: Eine Funktion, die im Code definiert ist, aber nicht aufgerufen wird. Diese Funktion sollte entfernt werden, um den Code übersichtlicher zu gestalten.

- Couplers: Diese Code Smells treten auf, wenn die Abhängigkeiten zwischen verschiedenen Teilen des Codes zu eng sind. Beispiele hierfür sind globale Variablen, die von vielen verschiedenen Klassen verwendet werden, oder enge Kopplung zwischen Klassen, die dazu führen, dass Änderungen an einer Klasse Auswirkungen auf viele andere Klassen haben. Gute Softwareentwicklung sollte eine lose Kopplung zwischen verschiedenen Teilen des Codes fördern. Beispiel: Eine Klasse, die eine globale Variable verwendet, die von vielen anderen Klassen im Code verwendet wird. Die Verwendung von globalen Variablen sollte vermieden werden, um den Code besser zu strukturieren und zu isolieren.

2. Why Software Architectur matters?

1. Welche Anforderungen werden von unterschiedlicher Seite an die SW gestellt?

    - Funktionale Anforderungen: Diese beschreiben, welche Funktionen und Aufgaben die Software erfüllen soll. Beispiele sind die Verarbeitung von Daten, die Speicherung von Informationen oder die Kommunikation mit anderen Systemen.

    - Nicht-funktionale Anforderungen: Diese beziehen sich nicht direkt auf die Funktionalität der Software, sondern auf ihre Eigenschaften, wie Leistung, Zuverlässigkeit, Sicherheit und Benutzerfreundlichkeit.

2. Welche Qualitätsattribute für eine Software gibt es und was bedeuten sie?

    - Funktionalität: Die Fähigkeit der Software, die vom Benutzer erwarteten Funktionen und Aufgaben auszuführen.

    - Zuverlässigkeit: Die Fähigkeit der Software, in einem bestimmten Zeitraum fehlerfrei zu funktionieren.

    - Leistung: Die Geschwindigkeit und Effizienz, mit der die Software Aufgaben ausführt und Ressourcen verwendet.

    - Benutzerfreundlichkeit: Die Einfachheit, Klarheit und Effektivität der Benutzeroberfläche und der Interaktion mit der Software.

    - Wartbarkeit: Die Fähigkeit der Software, Änderungen oder Reparaturen durchzuführen und zu implementieren, ohne dass andere Teile der Software beeinträchtigt werden.

2.1. Fokus auf die Priorität von Qualitätsattributen in Abhängigkeit von den Usecases.

    - E-Commerce-Website: In diesem Fall wäre Funktionalität die höchste Priorität, da die Website in der Lage sein muss, Bestellungen zu verarbeiten und Zahlungen sicher abzuwickeln. Benutzerfreundlichkeit und Leistung wären ebenfalls wichtige Faktoren, um sicherzustellen, dass die Kunden die Website einfach und schnell nutzen können.

    - Finanzanwendungen: Bei Finanzanwendungen wie Online-Banking-Apps oder Investitionsplattformen ist Sicherheit das höchste Qualitätsattribut, um die Privatsphäre und Sicherheit der Nutzerdaten zu gewährleisten. Zuverlässigkeit ist auch ein wichtiger Faktor, da fehlerhafte Anwendungen schwerwiegende finanzielle Auswirkungen haben können.

3. Was sind die wichtigsten 3 SW - Architekturen gereiht nach deren historischen Entwicklung?

    - Monolithische Architektur: Monolithische Architekturen waren die erste Architektur, die für die Entwicklung von Softwareanwendungen verwendet wurde. Sie basieren auf einem einzigen, in sich geschlossenen Code, der alle Funktionen und Module der Anwendung enthält. Alle Teile der Anwendung teilen sich dieselbe Datenbank und dieselben Ressourcen. Das bedeutet, dass Änderungen an einem Teil der Anwendung oft Änderungen an anderen Teilen erfordern.

- Vor- und Nachteile der monolithischen Architektur:

- Vorteile:

Einfache Entwicklung und Wartung, da alles in einer einzigen Anwendung zusammengefasst ist. Geringere Komplexität, da es nur eine Anwendung gibt. Bessere Performance, da keine Netzwerklatenz zwischen den verschiedenen Teilen der Anwendung auftritt.

- Nachteile:

Skalierbarkeit kann schwierig sein, da die gesamte Anwendung skaliert werden muss, auch wenn nur ein Teil der Anwendung eine hohe Belastung aufweist. Begrenzte Technologieauswahl, da alle Teile der Anwendung dieselbe Technologie verwenden müssen. Wartbarkeit kann schwierig sein, da alle Teile der Anwendung eng miteinander verbunden sind und Änderungen an einem Teil der Anwendung oft Änderungen an anderen Teilen erfordern. Anwendungsbereiche/Beispiele: Kleinere bis mittlere Anwendungen, die keine komplexe Skalierung benötigen, wie Blogs, Webseiten, Content-Management-Systeme (CMS).

- Client-Server-Architektur: Bei der Client-Server-Architektur ist die Anwendung in zwei Teile unterteilt: einen Client-Teil und einen Server-Teil. Der Client-Teil ist für die Interaktion mit dem Benutzer verantwortlich und der Server-Teil für die Verarbeitung und Speicherung von Daten. Die Kommunikation zwischen dem Client und dem Server erfolgt über das Netzwerk.

- Vor- und Nachteile der Client-Server-Architektur:

- Vorteile:

Einfache Skalierbarkeit, da der Server-Teil der Anwendung skalierbar ist, ohne dass der Client-Teil geändert werden muss. Flexibilität bei der Auswahl von Technologien, da der Client und der Server unterschiedliche

Technologien verwenden können. Verbesserte Sicherheit, da die meisten kritischen Daten auf dem Server gespeichert sind und der Client nur Zugriff auf die notwendigen Daten hat. Nachteile:

Erhöhte Netzwerklatenz, da alle Anfragen und Antworten über das Netzwerk übertragen werden müssen. Komplexere Entwicklung, da die Anwendung in zwei Teile aufgeteilt ist und die Kommunikation zwischen beiden Teilen verwaltet werden muss. Höhere Kosten, da der Server-Teil der Anwendung häufig auf leistungsstarken Servern ausgeführt wird. Anwendungsbereiche/Beispiele: Unternehmen, die eine zentrale Datenbank verwalten müssen, wie Banken, E-Commerce-Websites, soziale Netzwerke.

- Microservices-Architektur: Microservices sind eine moderne Architektur, bei der die Anwendung in mehrere kleine Dienste aufgeteilt wird, die unabhängig voneinander entwickelt und bereitgestellt werden können. Jeder Dienst erfüllt eine spezifische Funktion und kann von anderen Diensten aufgerufen werden. Die Kommunikation zwischen den Diensten erfolgt über Netzwerkprotokolle.
- Vor- und Nachteile der Microservices-Architektur:
- Vorteile:

Hohe Skalierbarkeit, da jeder Dienst unabhängig skaliert werden kann. Flexibilität bei der Auswahl von Technologien, da jeder Dienst unabhängig entwickelt werden kann. Bessere Wartbarkeit, da Änderungen an einem Dienst nicht die gesamte Anwendung betreffen. Bessere Ausfallsicherheit, da ein Ausfall eines Dienstes die gesamte Anwendung nicht zum Absturz bringen wird.

- Nachteile:

Erhöhte Komplexität, da die Anwendung in viele unabhängige Dienste aufgeteilt ist und die Kommunikation zwischen ihnen verwaltet werden muss. Erhöhter Ressourcenbedarf, da jeder Dienst seine eigene Datenbank und Ressourcen benötigt. Schwierigere Integrationstests, da die Anwendung aus vielen unabhängigen Diensten besteht. Anwendungsbereiche/Beispiele: Große und komplexe Anwendungen, die eine hohe Skalierbarkeit und Flexibilität erfordern, wie Online-Marktplätze, Streaming-Dienste, Cloud-Anwendungen.

Es ist wichtig zu beachten, dass es auch viele weitere Architekturen gibt, die je nach Anforderungen und Bedürfnissen der Anwendung eingesetzt werden können.

3. Tiers vs. Layers: Worin besteht der Unterschied zwischen "Tiers" und "Layers" im Zusammenhang mit SW - Architektur (SOF)

- In der Software-Architektur beziehen sich "Tiers" und "Layers" auf verschiedene Konzepte der Strukturierung von Softwareanwendungen.
    - Tiers (Schichten) beziehen sich auf die Unterteilung einer Anwendung in separate physische Schichten, die auf verschiedenen physischen Servern ausgeführt werden können. Jede Schicht hat spezifische Funktionen und Verantwortlichkeiten und kommuniziert mit den anderen Schichten über definierte Schnittstellen. Die häufigsten Schichten in einer Anwendung sind die Präsentationsschicht (User Interface), die Anwendungslogik-Schicht und die Datenzugriffsschicht.
    - Im Gegensatz dazu bezieht sich "Layers" (Ebenen) auf die Unterteilung einer Anwendung in separate logische Schichten innerhalb eines einzelnen physischen Servers. Jede Schicht hat spezifische Funktionen und Verantwortlichkeiten und kommuniziert mit den anderen Schichten

über definierte Schnittstellen. Die häufigsten Schichten in einer Anwendung sind die Präsentationsschicht (User Interface), die Geschäftslogik-Schicht und die Datenzugriffsschicht.

- Der wesentliche Unterschied zwischen Tiers und Layers besteht also darin, dass Tiers die physische Struktur einer Anwendung auf verschiedenen Servern widerspiegeln, während Layers die logische Struktur einer Anwendung innerhalb eines einzigen physischen Servers widerspiegeln.

In der Praxis werden Tiers oft verwendet, um eine Anwendung zu skalieren und eine höhere Verfügbarkeit zu gewährleisten, während Layers häufig verwendet werden, um die Struktur einer Anwendung zu vereinfachen und die Wartbarkeit zu verbessern.

4 Application Architectures

1.1 Erkläre die Unterschiede zwischen Server-based, Client/Server und Client-based Architecuture mit Beispielen, sowie deren Vor- und Nachteile.

- Application Architectures beschreiben die Struktur und Organisation von Software-Anwendungen und wie ihre verschiedenen Komponenten zusammenarbeiten, um Geschäftsprozesse zu unterstützen und IT-Ziele zu erreichen. Im Folgenden erläutern wir die Unterschiede zwischen den Architekturtypen Server-basiert, Client/Server und Client-basiert.

1.2 Server-based Architecture: Bei einer serverbasierten Architektur liegt die gesamte Anwendung auf dem Server, während die Clients (z. B. Browser oder mobile Geräte) lediglich als Schnittstelle zur Anwendung dienen. Ein Beispiel hierfür wäre eine Webanwendung, bei der der Server alle Anwendungslogik und Datenverarbeitung übernimmt, während der Client nur zur Anzeige von Informationen und zur Eingabe von Benutzerdaten verwendet wird.

- Vorteile:

Geringere Anforderungen an die Client-Hardware, da die Anwendung auf dem Server ausgeführt wird.

Zentralisierte Verwaltung der Anwendung und Daten.

Einfache Skalierbarkeit durch Hinzufügen weiterer Server.

- Nachteile:

Höhere Netzwerklatenz aufgrund der Notwendigkeit, Anfragen an den Server zu senden und Antworten zu erhalten.

Eingeschränkte Funktionalität, da alle Verarbeitungsschritte auf dem Server ausgeführt werden müssen.

Erhöhte Abhängigkeit vom Server, da die Anwendung ohne Verbindung zum Server nicht funktioniert.

1.3 Client/Server Architecture:

Eine Client/Server-Architektur teilt die Anwendung in zwei separate Komponenten auf: einen Client, der die Benutzeroberfläche und die Präsentationsebene enthält, und einen Server, der die Anwendungslogik und die Datenverarbeitung ausführt. Ein Beispiel für eine Client/Server-Architektur ist eine Desktopanwendung, bei der der Client auf dem Desktop des Benutzers ausgeführt wird und der Server für die Verarbeitung von Daten und die Überprüfung von Berechtigungen zuständig ist.

- Vorteile:

Bessere Skalierbarkeit, da der Server unabhängig vom Client skaliert werden kann.

Mehrere Clients können gleichzeitig auf die Anwendung zugreifen.

Verbesserte Sicherheit, da kritische Daten auf dem Server gespeichert sind.

- Nachteile:

Komplexere Architektur, da die Anwendung in zwei separate Komponenten aufgeteilt wird.

Höhere Kosten durch die Notwendigkeit, den Server zu warten und zu skalieren.

Höhere Anforderungen an die Client-Hardware im Vergleich zu serverbasierten Architekturen.

1.3 Client-based Architecture:

Bei einer Client-basierten Architektur ist die gesamte Anwendung auf dem Client-Gerät installiert. Ein Beispiel für eine Client-basierte Architektur ist eine mobile App, bei der alle Anwendungslogik und Datenverarbeitung auf dem mobilen Gerät des Benutzers ausgeführt werden.

- Vorteile:

Geringere Netzwerklatenz, da keine Kommunikation mit einem Server erforderlich ist.

Volle Kontrolle über die Anwendung und die Daten auf dem Client-Gerät.

Bessere Leistung, da die Anwendung auf dem Client-Gerät ausgeführt wird.

- Nachteile:

Schwierigere Skalierbarkeit, da alle Verarbeitungsschritte auf dem Client-Gerät ausgeführt werden.

2.1 Was versteht man unter den folgenden Begriffen:

- Middleware: Eine Software-Schicht, die zwischen Anwendungen und dem Betriebssystem arbeitet und die Interoperabilität zwischen verschiedenen Systemen ermöglicht.
- Multi-tier Architecture: Eine Architektur, die eine Anwendung in mehrere logische Schichten aufteilt, um die Skalierbarkeit, Wartbarkeit und Sicherheit zu verbessern.
- API: Eine Programmierschnittstelle, die die Kommunikation zwischen verschiedenen Anwendungen oder Komponenten erleichtert. Integration Platform: Eine Plattform, die es ermöglicht, verschiedene Anwendungen und Systeme miteinander zu integrieren und zu automatisieren.
- SW - Stack: Eine Sammlung von Software-Komponenten, die zusammenarbeiten, um eine bestimmte Anwendung oder Systemfunktion zu erfüllen.
- Virtualization: Eine Technologie, die es ermöglicht, mehrere virtuelle Betriebssysteme auf einem einzigen physischen Computer auszuführen.
- Container: Eine virtuelle Umgebung, die es ermöglicht, Anwendungen und ihre Abhängigkeiten isoliert voneinander auszuführen.

- Micro Services: Eine Architektur, bei der eine Anwendung in mehrere unabhängige, lose gekoppelte Dienste aufgeteilt wird. Jeder Dienst erfüllt eine bestimmte Funktion und kann unabhängig voneinander entwickelt und skaliert werden.