3.1 Rätselhafte Bezeichnung: Ein häufiges Problem in der Programmierung ist die Wahl unklarer oder irreführender Bezeichner für Funktionen, Variablen und Klassen. Entwickler zögern oft, Code umzubenennen, obwohl aussagekräftige Namen langfristig Zeit sparen und Missverständnisse vermeiden. Unklar benannte Teile deuten häufig auf tiefere Designfehler hin, die durch Umbenennen behoben werden können.

Problem: Unklare und irreführende Bezeichner. **Lösung:** Umbenennung und präzise Wahl von Namen, um den Zweck von Code klar und verständlich zu machen.

3.2 Redundanter Code: Redundanter Code tritt auf, wenn gleiche oder ähnliche Codeabschnitte mehrfach vorkommen. Dies erhöht das Risiko von Fehlern und macht Wartung und Änderungen kompliziert. Wiederholung kann durch Refactoring wie "Funktion extrahieren" oder "Methode nach oben verschieben" reduziert werden.

Problem: Mehrfacher Code, der schwer zu ändern und anfällig für Fehler ist. **Lösung:** Verwendung von Refactoring-Techniken, um redundante Codeabschnitte zu extrahieren und zu vereinheitlichen.

3.3 Lange Funktion: Lange Funktionen sind schwer verständlich und erschweren die Wartbarkeit. Kurze, prägnante Funktionen sind dagegen klarer, wiederverwendbar und erleichtern die Lesbarkeit. Lange Funktionen sollten in kleinere, gut benannte Funktionen unterteilt werden.

Problem: Lange, schwer verständliche Funktionen. **Lösung:** Aufteilung langer Funktionen in kleinere, gut benannte und übersichtliche Funktionen.

3.4 Lange Parameterliste: Lange Parameterlisten führen oft zu Verwirrung und schwieriger Wartung. Zu viele Parameter können durch die Einführung von Objekten, Abfragen oder anderen Techniken wie "Parameterobjekt" oder "Vollständiges Objekt erhalten" reduziert werden.

Problem: Lange und unübersichtliche Parameterlisten. **Lösung:** Reduzierung der Parameterliste durch Umstrukturierung von Code und Verwendung geeigneter Datenstrukturen.

3.5 Globale Daten: Globale Daten, wie Variablen und Klassen- oder Singleton-Daten, sind schwer zu verwalten und führen häufig zu Bugs, da sie von überall im Code geändert werden können. Eine Lösung besteht darin, diese Daten zu kapseln und den Zugriff auf sie zu kontrollieren.

Problem: Verwendung von globalen Daten, die schwer zu kontrollieren sind und Fehler verursachen. **Lösung:** Kapselung globaler Daten und Begrenzung des Zugriffs auf die betroffenen Bereiche, um Änderungen nachvollziehbar und kontrollierbar zu machen.

3.6 Veränderliche Daten: Änderungen an Daten können unerwartete Fehler verursachen, besonders wenn sie seltene Umstände betreffen. Eine Lösung ist, Daten unveränderlich zu gestalten und bei Änderungen eine Kopie der Datenstruktur zu erstellen. Es gibt Techniken wie das Kapseln von Variablen, das Teilen von Variablen und das Trennen von Abfragen und Änderungen, um Risiken durch veränderliche Daten zu minimieren.

3.7 Divergierende Änderungen: Divergierende Änderungen entstehen, wenn ein Modul aus verschiedenen Gründen häufig geändert wird. Eine Lösung ist, separate Module für unterschiedliche Kontexte (z. B. Datenbankzugriff und Finanzverwaltung) zu erstellen, um die Änderungen einfacher und überschaubarer zu gestalten.

3.8 Chirurgie mit der Schrotflinte: Dieser Code-Smell tritt auf, wenn viele kleine Änderungen an verschiedenen Stellen des Codes vorgenommen werden müssen. Eine Lösung ist, Funktionen und Felder zu verschieben oder zu extrahieren, um Änderungen zu bündeln und die Modifikationen an einer zentralen Stelle vorzunehmen.

3.9 Feature-Neid: Feature-Neid entsteht, wenn eine Funktion mehr mit den Daten und Funktionen eines anderen Moduls interagiert als mit denen ihres eigenen Moduls. Eine Lösung ist, die Funktion in das passende Modul zu verschieben. Wenn eine Funktion mehrere Module betrifft, sollte sie in das Modul mit den meisten relevanten Daten verschoben werden.

3.10 Datenklumpen: Wenn mehrere Datenelemente immer zusammen verwendet werden, sollten sie zu einem Objekt zusammengefasst werden, um den Code übersichtlicher zu gestalten. Eine Lösung ist, Klassen zu erstellen, anstatt nur einfache Datensätze zu verwenden, und so die Methodensignaturen zu vereinfachen und unnötige Parameter zu reduzieren.

3.11 Obsession für elementare Datentypen: Der übermäßige Gebrauch von primitiven Datentypen kann dazu führen, dass das Verhalten nicht gut gekapselt ist. Eine Lösung besteht darin, diese primitiven Datentypen in passende Klassen zu überführen, um das Verhalten besser zu kapseln und den Code klarer zu strukturieren.

3.12 Wiederholte switch-Anweisungen: Wiederholte switch- oder if/else-Anweisungen, die an mehreren Stellen im Code vorkommen, verursachen Redundanz und machen spätere Änderungen schwieriger. Polymorphie kann helfen, diese Redundanzen zu vermeiden, indem die Logik zentralisiert und eleganter gestaltet wird.

3.13 Schleifen: Schleifen sind veraltet und können durch modernere Konzepte wie Pipelines, Filter und Map ersetzt werden. Diese Techniken ermöglichen eine klarere und funktionalere Handhabung von Daten.

3.14 Träges Element: Manchmal sind bestimmte Programmierelemente überflüssig, z.B. Funktionen oder Klassen, die nie wachsen oder oft genutzt werden. Diese sollten entweder inline platziert oder entfernt werden, um den Code zu bereinigen.

3.15 Spekulative Generalisierung: Dieser Code-Smell tritt auf, wenn Entwickler unnötige Erweiterungen oder Spezialfälle hinzufügen, die derzeit nicht gebraucht werden. Dies führt zu

unübersichtlichem und wartungsintensivem Code. Es sollte darauf geachtet werden, nur das zu implementieren, was wirklich erforderlich ist.

3.16 Temporäres Feld: Ein Feld, das nur unter bestimmten Bedingungen einen Wert bekommt, kann zu Verwirrung führen. Eine Lösung ist, das Feld in eine eigene Klasse auszulagern und den zugehörigen Code dort zu verlagern.

3.17 Mitteilungsketten: Eine Mitteilungskette entsteht, wenn ein Objekt ein weiteres nach einem anderen fragt. Das führt zu einer engen Kopplung der Objekte. Dies kann durch Delegation und das Verschieben von Funktionen gelöst werden, um die Struktur flexibler und weniger anfällig für Änderungen zu machen.

3.18 Vermittler: Zu viel Delegation kann den Code unnötig komplex machen. Wenn eine Klasse nur Delegationen an andere Klassen vornimmt, kann es sinnvoll sein, den Vermittler zu entfernen und direkt mit den relevanten Objekten zu arbeiten.

3.19 Insiderhandel

Softwareentwickler neigen dazu, hohe Barrieren zwischen ihren Modulen zu errichten, wobei sie sich über den Austausch von Daten beklagen, da er die Kopplung verstärkt. Es ist jedoch wichtig, dass dieser Austausch auf ein Minimum beschränkt wird, indem wir sorgfältig vorgehen. Um die Kommunikation zwischen Modulen zu reduzieren, können Funktionen und Felder verschoben werden. Wenn Module gemeinsame Interessen haben, kann ein drittes Modul erstellt werden, um diese zu vereinen, oder Delegation verbergen verwendet werden, um einen Vermittler zu schaffen. Vererbung führt oft zu heimlichen Absprachen zwischen Unterklassen und Basisklassen, was durch Delegation ersetzt werden kann, wenn nötig.

3.20 Umfangreiche Klasse

Klassen, die zu viele Aufgaben übernehmen und viele Felder besitzen, neigen zu redundanten Code. Ein Ansatz zur Lösung dieses Problems ist, Klassen zu extrahieren, indem mehrere verwandte Variablen in eine eigene Komponente zusammengefasst werden. Wenn eine Klasse viele Felder besitzt, aber nicht alle verwendet, kann eine Extraktion sinnvoll sein. Ebenso sollte redundant wiederholter Code reduziert werden, indem Methoden aufgeteilt und der Code vereinfacht wird. Wenn bestimmte Funktionalitäten nur von einem Teil der Clients benötigt werden, sollte die Klasse extrahiert oder aufgeteilt werden.

3.21 Alternative Klassen mit unterschiedlichen Schnittstellen

Die Substituierbarkeit von Klassen ist von Vorteil, aber nur, wenn ihre Schnittstellen übereinstimmen. Wenn das nicht der Fall ist, kann die Funktionsdeklaration geändert werden, um sie anzupassen. Bei Bedarf kann das Verhalten in Klassen verschoben werden, um die Schnittstellen weiter anzugleichen, und Redundanzen durch das Extrahieren von Basisklassen reduziert werden.

3.22 Datenklasse

Datenklassen, die nur Felder sowie Getter- und Setter-Methoden enthalten, sind oft einfache Datencontainer und werden in der Regel von anderen Klassen in viel zu großem Ausmaß

verändert. Um dieses Problem zu beheben, sollten Getter und Setter entfernt und die Felder gekapselt werden. Wenn möglich, sollte das Verhalten in die Datenklasse verschoben werden, andernfalls kann eine Funktion extrahiert werden. Es gibt jedoch Ausnahmen, insbesondere wenn die Datenstruktur unveränderlich ist, was bedeutet, dass die Felder nicht gekapselt werden müssen.

3.23 Ausgeschlagenes Erbe

Wenn Unterklassen das Erbe ihrer Basisklasse nicht benötigen, deutet dies auf eine falsche Hierarchie hin. Die Lösung besteht darin, die überschüssigen Teile in eine Geschwisterklasse zu verschieben und die Basisklasse nur mit den gemeinsamen Merkmalen zu versehen. In manchen Fällen ist es jedoch unproblematisch, Unterklassen zu verwenden, solange die Schnittstelle der Basisklasse noch unterstützt wird. Wenn jedoch die Schnittstelle nicht unterstützt wird, sollte die Hierarchie durch Delegation ersetzt werden.

3.24 Kommentare

Kommentare sind oft hilfreich, aber sie sollten nicht als "Deodorant" für schlechten Code verwendet werden. Wenn der Code klar und gut strukturiert ist, sind Kommentare oft überflüssig. Falls doch ein Kommentar notwendig ist, kann es hilfreich sein, das Verhalten durch Funktionsextraktion zu verlagern oder die Methode umzubenennen. Kommentare können auch dazu verwendet werden, Unsicherheiten oder die Gründe für bestimmte Entscheidungen zu dokumentieren, um zukünftigen Entwicklern zu helfen.