Einsatz von Design Patterns

Designpatterns sind generalisierte komplexe Programmstrukturen, die sprachenunabhängig (also abstrakt) modelliert werden.

Der konsequente Einsatz solcher Strukturen vereinfacht die Dokumentation und Kommunikation innerhalb eines Entwicklerteams, da gängige Programmstrukturen bereits dokumentiert sind und in Folge einfach verstanden und repliziert werden können. Ein einmal entwickelter Designpattern reflektiert die Erfahrungen eines Entwicklungsteams und kann nach dem mehrmaligen Einsatz in der Praxis als Wissensgrundlage verwendet werden. Dadurch bieten solche etablierten Konstrukte für Entwickler, die mit dem bestehenden Projekt nicht vertraut sind, eine Wissensbasis, welche die Einarbeitungszeit verkürzt.

Das Standardwerk "Design Patterns: Elements of Reusable Object-Oriented Software," (von Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Publising Company, 1995) definiert Designpatterns als:

„… descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

Die vier Autoren (die so genannte „Gang of Four") legten zum ersten Mal fest, welche Attribute ein Softwarepattern beinhalten sollte. Danach definierten Sie die 23 Patterns, die als Basis der darauf folgenden Patterns gesehen werden können. Die 23 Grundpatterns werden in der Literatur auch als „GoF Patterns" bezeichnet. Jedes solche Konstrukt wird von der „GoF" über die vier Eigenschaften: Patternname, Problem, Lösung und Konsequenzen definiert.

1. Die klassischen Patterns der „Gang of Four"

Die 23 Basispatterns, die von der „Gang of Four" definiert wurden, unterteilen sich in drei Gruppen:

- **Creational Patterns** - beziehen sich auf Objektinstanzierung
- **Structural Patterns** - beziehen sich auf Objektbeziehungen
- **Behavioral Patterns** - beziehen sich auf Objektinteraktion

Im Folgenden beschreiben wir diese drei Gruppen etwas genauer und liefern jeweils einige Beispiele von Patterns, die diesen Gruppen angehören.

1.1. Creational Patterns

Creational Patterns übernehmen die Aufgabe der Objektinstanzierung. Dabei bleibt dem Benutzer solcher Patterns verborgen wie und welches Objekt erzeugt worden ist. Dadurch werden klare abstrakte Interfaces definiert, die sich durch die Vererbungshierarchie ergeben.

Beispiel: Pattern „Abstract Factory“

Adressiertes Problem:
Verschiedene Klassen eines Ableitungsbaumes sollen anhand von Parametern instanziert und der Nutzerklasse übergeben werden.

Lösung:
Die 'Abstract Factory' Klasse generiert z.B. anhand eines Strings eine Instanz und gibt die Basisklasse an die aufrufende Instanz zurück. Dabei sollte die Basisklasse alle vom Client benötigten Interfaces beinhalten. Ein klassisches Beispiel ist das Java Swing LookandFeel. Hier wird anhand eines String die richtige Implementierung zurückgegeben. Dabei bleibt die Implementierung den Nutzerklassen (z.B. das Layout der Buttons) verborgen.

Konsequenzen:
- Klare Entkoppelung der verschiedenen dem Nutzer zurückgegeben Objekte.
- Das Hinzufügen oder das Ändern der abgeleiteten Klassen erfordert nur eine Kompilierung der 'Abstract Factories' und der geänderten Komponenten.

Beispiel: Pattern „Singleton“

Adressiertes Problem:
Da die Initialisierung von Database Connections sehr kostspielig ist, sollten diese minimiert werden; dabei soll man die Klasse immer noch vererben können.

Lösung:
Der Singleton Pattern erfüllt diese Aufgabe, indem er die Instanzierung in einer eigenen Methode kapselt. Diese Lösung kann mit Hilfe von statischen Variablen gelöst werden. Singleton beinhaltet eine private Klassenvariable des gleichen Typs. Wenn nun eine Instanz verlangt wird, wird beim ersten Mal das Objekt instanziert; ab dann wird immer dasselbe Objekt returniert. Probleme ergeben sich beim Subclassing. Auch da muss die Instanzierung über dieses statische Objekt getätigt werden, damit sichergestellt ist, dass nur ein Objekt existiert. Dies wird dadurch gelöst, dass man die Instanzierungsmethode überlädt und damit das abgeleitete Objekt kreiert wird.

Konsequenzen:
- Clients haben kein Wissen darüber, wie oft das Objekt instanziert worden ist; die Kontrolle obliegt dem Singleton.
- Erweiterung auf eine überschaubare Menge an Instanzen leicht möglich.
- Die Verwaltung von applikationsweiten Singletons kann über ein Register gehandhabt werden. Dadurch gibt es für alle Nutzer einen zentralen Einstiegspunkt.

1.2. Structural Patterns

Structural Patterns beschreiben die Zusammensetzung und die Zusammenlegung einzelner Strukturen. Dabei kann man zwischen Objekt- und Klassenmustern unterscheiden. Objektmuster bilden Objektstrukturen, die zum einen Objekte beinhalten und zum anderen Objekte zu gemeinsamen Konstrukten zusammenfassen. Klassenmuster verwenden die Vererbung als Möglichkeit um Schnittstellen zu anderen Klassen zu definieren.

Beispiel: Pattern „Adaptor“

Adressiertes Problem:
Zwei Klassen, die nicht direkt im Zusammenhang stehen müssen, sollen verbunden werden, damit Funktionalitäten wiederbenützt werden können, wobei die zu benutzende Klasse aus verschiedenen Gründen (z.B. ist Teil einer anderen API) nicht verändert werden darf.

Lösung:
Es wird innerhalb des Adaptors die verwendbare Klasse als Instanz benutzt. Alle Interaktionen werden dann in die dem benutzten Objekt verständlichen Aufrufe umgeleitet. Dadurch wird die Funktionalität der Klasse ummantelt. Dabei ist es auch möglich mehrere Objektinstanzen zu verwenden.
Eine zweite Variante ist, die zu verwendende Klasse abzuleiten. Dies macht nur dann Sinn, wenn Methoden der zu adaptierenden Klasse überschrieben werden müssen.

Konsequenzen:
Bei der Verwendung von Objektinstanzen ist das Überschreiben von Methoden nur über das Subklassen des verwendeten Objektes möglich. Daher ist eher die Klassenvariante zu wählen.

1.3. Behavioral Patterns

Diese Patterns beschreiben die Interaktion zwischen den Objekten. Dabei unterscheidet man zwischen Klassen- und Objektpatterns.

Bei den Klassenpatterns werden diese Interaktionen über Vererbung gelöst. Dadurch lässt sich z.B. eine Algorithmusstruktur in den abstrakten Klassen darstellen. Die vererbte Klasse hat dann die Möglichkeit, abweichendes Verhalten zu implementieren, ohne dass der vorgegebene Algorithmus verändert wird (z.B. Template Pattern).

Objektpatterns gehen auf die Komposition mehrerer Objekte ein (die sich in Gruppen zusammenfassen lassen), die miteinander operieren können (z.B. Subjekt und Observer im Observer Pattern).

Beispiel: Pattern „Observer“

Adressiertes Problem:
Bei der Änderung der Eigenschaften eines Objektes sollen *n* Objekte sofort über die Änderung verständigt werden. Weiters soll sichergestellt sein, dass ein Hinzufügen eines weiteren Objektes, das verständigt werden will, keine Änderung an dem Informationsmechanismus bedeutet.

Lösung:
Das Subjekt der Überwachung (Observable in Java) implementiert die Möglichkeit eines Observers sich zu registrieren. Bei Änderungen im Subjekt werden dann alle angehängten Observer benachrichtigt.

Konsequenzen:
- Observer können Änderungen zu oft bekannt geben.
- Die Klasse die Überwachung kennt keine konkrete Klasse, sondern kennt die Objekte nur als Observerinterfaces.
- Da die Requests Broadcasts sind, wird keine Antwort von den Observern erwartet – dies muss bei der Fehlerbehandlung berücksichtigt werden.

2. Moderne Patterns aus einem J2EE-Environment

Bedingt durch ständige technologische Weiterentwicklungen entstehen natürlich auch neue Patterns. Diese bauen auf bereits bekannten auf, beziehungsweise integrieren diese in ihre Struktur. Zur Demonstration jüngerer Design Patterns behandelt der folgende Abschnitt einige Patterns aus dem J2EE-Umfeld.

Die J2EE-Spezifikation sieht eine 4-Schichten Trennung (Client, Presentation, Business und Enterprise Information System Tier) vor. Hier legen wir den Focus auf die Presentation Tier mit einer MVC-Implementierung. Der Front Controller und der View Helper bieten dafür einen geradlinigen Lösungsansatz.

Da die Businesslogik in einem J2EE-Environment über voneinander getrennte (also verteile) Systeme gelöst ist, muss die Anbindung der Presentationsschicht von der Businessschicht entkoppelt werden. Dadurch haben Änderungen an der Kommunikation keinen Einfluss auf den Presentation Tier. Damit beschäftigen sich die letzten beiden Patterns.

Beispiel: Pattern „Front Controller“

Adressiertes Problem:
Applikationsweite anfragenunabhängige Prüfungen sollen nur einmal implementiert werden (z.B. Sessionmanagement). Dabei ist darauf zu achten, dass die Java Server Pages (JSP) unabhängig bleiben. Das Hinzufügen von programmierlogischen Komponenten sollte keine Auswirkungen auf die Views (JSP) haben. Als weitere Anforderung sollte die Vernetzung der einzelnen Seiten an zentraler Stelle vom Front Controller verwaltet werden können.

Lösung:
Der Lösungsansatz wäre, eine Controller Klasse zu definieren, die alle Anfragen der Views annimmt, die allgemeinen Prüfungen durchführt und dann die Anfragen an das Modell (Businesslogik) weiterleitet. Eine solche Lösung bietet das „Jakarta Struts Project“. Im so genannten ActionServlet können Prüfungen wie z.B. Logging und andere anfragenunabhängige Programmteile implementiert werden. Die Weiterleitung an das richtige Modell übernimmt ein XML-File. Die Businesslogik bearbeitet dann die Anfrage und übergibt nun dem ActionServlet die Rücksprungadresse mit der korrekten View.

Konsequenzen:
- Entkoppelung der JSPs von allgemeiner Businesslogik.
- Einfache Möglichkeit, applikationsweite und anfragenunabhängige Businesslogik zu implementieren.
- Es liegt ein Single Point of Failure vor.
- Einige Teile (z.B. Sessionmanagement) können projektunabhängig implementiert und so wiederverwendet werden.
- Änderungen der URLs brauchen nur in einem File getätigt werden und nicht in jeder einzelnen JSP Seite.

Beispiel: Pattern „View Helper“

Adressiertes Problem:
Die Businesslogik gibt Ergebnisse zurück, die dem Benutzer als HTML-View sichtbar gemacht werden. Generell neigen solche Views dazu, HTML mit eingebetteter Logik (Scriptlets) zu vermischen. Das macht HTML-Seiten unübersichtlich und erschwert das Arbeiten eines Webdesigners. Außerdem verleitet es dazu, dass der Entwickler diese Arbeiten durchführt, was in der Regel zu Zuständigkeitskonflikten führen kann.

Lösung:
Jeder Seite werden View Helper Beans zur Verfügung gestellt. Diese beinhalten keine Businesslogik, sondern dienen rein zur Darstellung. Mit Hilfe der Custom Tags wird auf diese Beans zugegriffen, ohne den HTML-Code zu verunstalten. Dadurch werden die Scriptlets aus den HTML-Seiten in die Taglibs ausgelagert und der Webdesigner hat wieder „reines“ HTML – die Logik dahinter bleibt ihm verborgen. Moderne HTML-Editoren unterstützen schon diese Funktionalität (z.B. Macromedia Dreamweaver MX 2004).

Konsequenzen:
- Minimierung von Scriptlets im HTML-Code.
- Klarere Trennung der Aufgaben des Webdesigners und des Entwicklers.
- View Helper Javabeans können nur einmal verwendet werden.

Beispiel: Pattern „Service Locater“

Adressiertes Problem:
In J2EE werden verschiedenste Services (Datenbankverbindung, Enterprise Java Beans (EJBs)) in einem Java Naming and Directory Interface (JNDI) abgelegt. Um z.B. ein EJB zu holen, muss die Präsentationsschicht die JNDI-Initiierung durchführen und kann erst dann auf das gesuchte Objekt zugreifen. Dieser Zugriff ist zeitaufwendig und sollte minimiert werden.

Lösung:
Der Service Locater sieht eine Implementierung eines Singelton Patterns vor.
In diesem werden alle Zugriffe auf das JNDI implementiert. Dadurch bleibt die technische Implementierung eines JNDI-Aufrufes vor der Präsentationsschicht gekapselt. Weiters wird gewährleistet, dass die Netzwerklast möglichst gering gehalten wird, da die Initiierung nur einmal durchgeführt wird. Wenn jetzt das Client Objekt nach einem EJB sucht, übernimmt der Service Locater die Kontrolle und setzt die Suchabfrage ab. Wenn der Vorgang erfolgreich war, wird das Ergebnis gecached, sodass eventuell gleiche Suchanfragen bereits aus dem Cache entnommen werden können.

Konsequenzen:
- Der Instanzierungsvorgang wird erheblich beschleunigt, da die Initialisierung des JNDI nur einmal vorgenommen wird.
- Jedes Suchresultat wird gecached, dadurch wird die Netzwerklast verringert.
- Die Zugrifflogik auf das JNDI-API wird gekapselt.

Beispiel: Pattern „Business Delegate“

Adressiertes Problem:
In einem EJB-Kontext erhält die Präsentationslogik nur über aufwändige APIs Zugriff auf die Businesslogik. Jede technische Änderung der darunter liegenden Kommunikationsschicht würde direkten Einfluss auf die implementierte Präsentationslogik nehmen, da keine klare Trennung zwischen Präsentations- und dem Kommunikationslayer liegt. Zudem muss die Präsentationslogik Exceptions abfangen, die ihr unbekannt sind (z.B. Netzwerkverbindung nicht verfügbar). Weiters wäre es wünschenswert, dass der Netzwerkverkehr minimal gehalten wird.

Lösung:
Alle Zugriffe auf die Businesslogik werden als Interfaces definiert, die dem Client vorliegen. Die dahinter liegende Implementierung der Zugriffslogik bleibt dadurch der Präsentationsschicht verborgen. Gibt es Änderungen, so bedeutet dies nicht automatisch eine Neukompilierung aller Komponenten, sondern eines der Patterns. Das Errorhandling für kommunikationsrelevante Fehler wird für die Präsentationsschicht in weiterverwendbare Fehlermeldungen konvertiert.

Konsequenzen:
- Die technische Implementierung befindet sich an zentraler Stelle.
- Die Präsentationsschicht hat klar definierte Schnittstellen zur Businesslogik
- Dem Entwickler wird vorgegaukelt, dass es sich um lokale Klassen handelt. Dies könnte zu einem Problem werden, wenn dieser zu oft und wahllos auf diese Klassen zugreift.
- Semaphoren können als Zugriffschutz auf die Businesslogik integriert werden.
- Die Businesszugriffe können im Business Delegate gecached werden. Dadurch wird der Netzwerkverkehr entlastet.

3. Abschließende Hinweise

Wie die hier vorgestellten Beispiele von Design Patterns zeigen, erleichtern sie den Prozess der Softwareentwicklung, indem sie bekannte Probleme abstrahieren und generalisieren. Design Patterns ermöglichen sowohl in der Analysephase als auch in der Implementierungsphase eine effizientere Softwareentwicklung.

Generell lässt sich festhalten: der Einsatz von Design Patterns rechnet sich umso mehr, je komplexer die jeweilige Aufgabenstellung ist.

Bei der Einführung von Design Patterns als methodisches Werkzeug sollte beachtet werden, dass eine gemeinsame Wissensbasis besteht. Damit wird verhindert, dass unterschiedliche Auffassungen von zu implementierenden Patterns zu unterschiedlichen Ergebnissen führen.

Ergänzend zu diesem Text steht Ihnen auf der InfraSoft Website ein Glossar zur Verfügung, in dem Sie die meisten der hier verwendeten Begriffe finden. Über das aktuelle Angebot an weiteren, kostenlosen Fachbeiträgen zur Softwareentwicklung informieren Sie sich bitte unter www.infrasoft.at/service.

Herbert Fresacher, B.Sc.
Wien, im Mai 2004

Der Autor ist Mitarbeiter der InfraSoft, einem Unternehmen, das auf komplexe Softwareentwicklungen spezialisiert ist. Die Experten der InfraSoft haben langjährige Erfahrungen in der Entwicklung und verfügen über fundierte Kenntnisse in Design, Analyse, Realisierung, Test und Projektmanagement.
Für **individuelle Beratungen** zur Entwicklung von Softwarelösungen und die Bereitstellung von **Realisierungsteams** wenden Sie sich bitte an info@infrasoft.at.