Dokumentation in der Software-Entwicklung

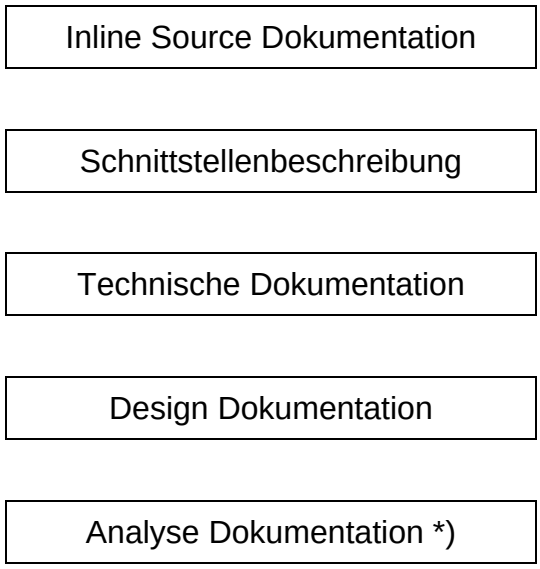
Dokumentation ist das Stiefkind der Softwareentwicklung. Obwohl der Wert professioneller und fundierter Dokumentation hinreichend bekannt ist, wird immer wieder gegen den Grundsatz verstoßen, dass Software ausreichend beschrieben werden muss.

Dabei ist Dokumentation ein Ansatz, der Softwareentwicklung mittel- und langfristig erst wirklich wirtschaftlich macht: Wartung sowie Weiterentwicklung der Software sind einfacher und bei der Durchführung von Neuerungen werden weniger Fehler eingeschleppt. Alles in allem wird die weitere Pflege der Software wesentlich kostengünstiger. Darüber hinaus steigt die Motivation der Programmierer, da sie durch das Beschreiben ihrer Arbeit im Rahmen der Dokumentation eine stärkere Vorstellung vom Wert ihrer Ergebnisse entwickeln. Das alles führt mittel- und langfristig zu gesteigerter Produktivität.

Dem gegenüber stehen leider oft kurzfristige Ressourcen-Beschränkungen. Einfach deshalb, weil die Entwicklung durch den Dokumentations-Prozess auf den ersten Blick länger dauert und das Mitführen der Dokumentation scheinbar keine produktive und damit eine kurzfristig entbehrliche Tätigkeit ist. Ein großer Irrtum! Wir möchten daher mit allem gebotenen Nachdruck auf die wirtschaftliche Bedeutung von umfassender Software-Dokumentation hinweisen und mit diesem Dokument die gängigsten Varianten in Erinnerung rufen.

1. Übersicht

Als Mindestanforderung an ein gut dokumentiertes Software-Projekt sind prinzipiell fünf Arten von Dokumentation zu unterscheiden:



*) gehört nicht zur Realisierungs-Phase und wird in diesem Dokument nicht näher behandelt

2. Inline Source Dokumentation

Unter *Inline Source Dokumentation* ist das Einfügen von Kommentaren direkt in den Programmcode zu verstehen. Dies geschieht je nach Programmiersprache mit unterschiedlichen Kommentarbefehlen, die es ermöglichen, in den Quellcode beliebigen Text einzubetten. Der kommentierte Text wird dann vom Interpreter oder Compiler ignoriert.

Bedeutung der Inline Source Dokumentation:

Für einen Programmierer, der den Code nicht geschrieben hat, ist oft aus der Codestruktur und den Variablen-, Funktions- und Klassennamen allein nicht ersichtlich, was das Funktionsprinzip des jeweiligen Codestückes ist. Es besteht somit die Gefahr, dass das Grundprinzip nicht verstanden wird und falsche Annahmen oder Vereinfachungen getroffen werden und in Folge ungünstige Änderungen durchgeführt werden.

Darüber hinaus wird der Programmierer gezwungen, sich Gedanken über das Funktionsprinzip seines Codes zu machen und dieses so einfach und kurz als möglich zu beschreiben. Das führt typischerweise dazu, dass unnötig komplizierte, ineffiziente oder anderweitig schlechte Algorithmen dem Programmierer schon während des Codierens auffallen. Die Idee ist also: „Code, den ich nicht kurz und bündig verständlich beschreiben kann, ist möglicherweise von vornherein schlecht".

Tipps zur Anwendung:

Wie schon erwähnt, sollte *Inline Source Dokumentation* möglichst kurz und bündig sein. Es ist vor allem darauf zu achten, dass nicht etwa die *Syntax* des Codes beschrieben wird (es ist z.B. völlig sinnlos vor einer Verzweigung einen Kommentar zu setzen, dass nun eine Verzweigung kommt - das geht aus den Schlüsselworten der Programmiersprache schon hinreichend hervor), sondern vielmehr die *Semantik* (also z.B. was bedeuten die Variablen; welche Realweltkonzepte werden mit einer bestimmten Klasse modelliert und wo sind die Einschränkungen des Modells) und das Funktionsprinzip (wurde ein in der Fachliteratur bekannter Algorithmus verwendet? Wenn ja, welcher, mit welchen Abweichungen?).

Ergänzend zu der Dokumentation in Form von Kommentaren sind auch die Namen von Variablen, Klassen, Objekten, Namesräumen und Funktionen als Teil der Dokumentation anzusehen und daher sinnvoll zu wählen. Funktionsnamen wie "func", "work" oder dgl. sind ebenso sinnlos wie Objektnamen "obj1", "obj2" oder Variablennamen "a", "b" etc. Wann immer möglich sollte ein Variablenname etwas über den Inhalt der Variable aussagen und nicht welchen Typs oder die wievielte Variable es ist.

Es gibt eine weithin bekannte Ausnahme von dieser Regel, die im Allgemeinen akzeptiert wird, und zwar die Bezeichnung von Ganzzahl-Schleifenvariablen. Sie werden üblicherweise mit i,j,k,... bezeichnet. Diese Konvention stammt noch aus FORTRAN Zeiten und hat sich soweit eingebürgert, dass sie heute als guter Programmierstil angesehen wird. Dennoch wäre es auch in diesen Fällen besser anzudeuten, welchen Inhalt die Variable hat.

Dazu ein Beispiel: der Programmierer möchte über eine 2-dimensionale Matrix (eine Tabelle aus Zahlen) eine Iteration durchführen:

```
// in C, C++, Java or C#
for( int i=0 ; i<iRowCount ; i++ )
{
	for( int j=0 ; j<iColCount; j++ )
	{
		result += matrix[i][j] ;
	}
}
```

Die Gefahr bei der Verwendung der eingebürgerten Bezeichnung i,j liegt nun darin, dass man, wenn man den inneren Code betrachtet, normalerweise keinen direkten Hinweis darauf hat, was Spalten und was Zeilen kennzeichnet. Eine Verwechslung von i und j ist schnell passiert und hat erst zur Laufzeit Folgen. Hier wäre die Bezeichnung iCol bzw. iRow wohl besser gewesen.

Daher gilt als abschließender Rat für Bezeichnungen, dass man sich immer eine der folgenden Fragen stellen sollte:

- Variablen: was beinhalten sie?
- Klassen: welches Realweltkonzept steckt dahinter?
- Funktionen: welche Arbeit verrichten sie?

Werden die Antworten auf diese Fragen in der Namensgebung berücksichtigt, so können die Namen selbst viel zur Lesbarkeit des Codes und damit seiner (Selbst-) Dokumentation beitragen.

3. Schnittstellenbeschreibung

Schnittstellenbeschreibungen sind Dokumente, welche Anzahl, Typen und erlaubte Reihenfolgen von Daten angeben, die zwischen Softwaremodulen ausgetauscht werden. Weiters sind darin Randbedingungen (d.h. Einschränkungen der erlaubten Werte) festgehalten. Das ist zum Beispiel: erlaubter Umfang (bei Daten in Listenform oder anderweitig großen Datenmengen - in erlaubten Datensätzen oder in maximal Byte gesamt); Art der erlaubten Antworten (i.e. Ausgaben) auf einen bestimmten Request (i.e. Eingabe); etc.

Bei den zu dokumentierenden Schnittstellen kann es sich um binäre Aufrufe handeln - in diesem Fall wird die Schnittstellenbeschreibung teilweise schon durch sog. Funktionsprototypen vom Programmierer geleistet. Dies ist jedoch niemals als Schnittstellenbeschreibung hinreichend, da zuwenig über die Semantik und erlaubte Kombinationen von Werten ausgesagt wird, sondern nur über die Typen der beteiligten Daten.

Oftmals handelt es sich bei den Schnittstellen um binäre oder Textbasierte Formate für Datenblöcke, die zwischen Computern oder Prozessen kommuniziert werden. Besonders bekannt ist in diesem Zusammenhang heutzutage XML. Auch hier gilt alles bisher Gesagte: soviel als möglich über den Inhalt (d.h. dessen Bedeutung) des Datenblocks schreiben bzw. über erlaubte Werte (in Kombination mit anderen Werten des Datenblocks). Im Falle von XML gibt es hierzu eine "automatisierte" Möglichkeit: DTD oder W3C Schema (XSL) Dokumente. Das sind Beschreibungen der Typen, der erlaubten Tags, der erlaubten Werte und vieles mehr, was zur Laufzeit von einem XML Parser verwendet werden kann, um zu validieren, dass ein erhaltenes XML Dokument der Spezifikation entspricht und zur "Programmierzeit" vom Programmierer gelesen werden kann, damit er eine genaue Vorstellung davon hat, was er schicken darf bzw. was er zu erwarten hat. Es gilt dennoch auch für XML Daten, dass eine manuelle Schnittstellenbeschreibung unumgänglich ist. Denn erstens ist ein XSL oder DTD Dokument schwer zu lesen (vor allem für Nicht-Programmierer) und zweitens sollte auch über die Semantik und die geplante Verarbeitung der so gesendeten Daten soviel als möglich ausgesagt werden. Der Sender der Daten muss nach dem Lesen des Dokuments eine klare Vorstellung davon haben, was mit seinen Daten passiert und was wahrscheinlich die Antwort sein wird - nicht nur, welche Daten erlaubt sind.

Bedeutung der Schnittstellenbeschreibung:

Es hat sich in der Softwareentwicklung immer wieder gezeigt, dass es vorkommt, dass die funktionale Aufteilung in Module und die Art der Daten, die zwischen den Modulen kommuniziert werden, nicht optimal ist. Das kann vielfältige Gründe haben: von einer verpatzten Designphase über unzureichendes Verständnis der beteiligten Programmierer oder gar des Fachbereichs bis hin zu Streitereien und Antipathien der beteiligten Parteien. Tatsache ist, dass viele dieser Fehler oder Ineffizienzen lange übersehen werden - spätestens bei einer manuellen, sauberen und umfangreichen Dokumentation der Schnittstellen fallen sie jedoch unweigerlich auf. Wenn diese Art der Dokumentation unterbleibt, ist die Gefahr sehr hoch, dass das System eine schlechte interne Kommunikation aufweist. Wie im Falle der *Inline Source Dokumentation* wird die Person, welche die *Schnittstellenbeschreibung* erstellt, gezwungen, sich nochmals alle Kommunikationsvorgänge zu vergegenwärtigen und bekommt so ein umfassendes Bild, in dem "Schönheitsfehler" herausstechen. Dies kann nicht genug betont werden - es handelt sich hierbei nicht um eine Idee aus dem Elfenbeinturm; die Wirksamkeit dieses "sich ein Bild von der Kommunikation machen" hat sich immer wieder in der Praxis massiv bestätigt.

Tipps zur Anwendung:

Über die äußere Form einer *Schnittstellenbeschreibung* lässt sich nun trefflich streiten - wir vertreten den Ansatz: lieber etwas, egal in welcher Form, als gar nichts. Wichtiger wäre hier, dass die Zeit erübrigt wird, um bei jeder Änderung die Dokumentation nachzuziehen (dies ist ein Aufruf an die Ressource Manager, nicht nur an die Programmierer!). Vorzugsweise sollte die Beschreibung ein einfach zu lesendes Format und evtl. auch leicht druckbares Format haben - nicht wenige Programmierer ziehen es vor, von Papier zu lesen.

4. Technische Dokumentation

Die *Technische Dokumentation* ist das Dokument bzw. die Dokumentensammlung, in der sich alles wieder finden sollte, was mit der speziellen Implementierung zu tun hat und nicht in der *Inline Source Dokumentation* oder der *Schnittstellen-beschreibung* zu finden ist. Sie sollte insbesondere die Funktionsprinzipien der Software oder des Moduls von einem höher stehenden Standpunkt als die *Inline Source Dokumentation* beschreiben. Das gilt auch und vor allem für die Einschränkungen der Funktionsfähigkeit. Wenn z.B. ein Algorithmus verwendet wurde, der bei einer bestimmten Art von Daten schlechte Performance erzielt, wäre dies hier zu dokumentieren. Die *Schnittstellenbeschreibung* kann auch als Teil dieser Dokumentation aufgefasst werden - es ist jedoch der Leserkreis ein etwas anderer: die Schnittstellendokumentation ist vor allem für den Programmierer interessant, während die technische Dokumentation sich etwa auch an den Administrator einer Software wendet, bzw. generell einen größeren Leserkreis hat.

Bedeutung der Technischen Dokumentation:

Ebenso wie die *Inline Source Dokumentation* und die *Schnittstellenbeschreibung* ist sie vital. Auch hier gilt: alles was nicht beschrieben ist, ist schlecht und wird sich eines Tages bemerkbar machen. Typischerweise rächen sich Versäumnisse dann als überlange Wartungsphasen oder noch schlimmer im Einführen von neuen Fehlern während der Wartung - dies, weil die Beteiligten kein vollständiges Bild über die Funktion der Software hatten.

Es wird zuweilen argumentiert, dass eine derart umfangreiche Dokumentation sinnlos ist, da die beteiligten Programmierer erstens das ganze Bild gar nicht benötigen und zweitens gar nicht alle Details im Kopf behalten könnten. Das ist nach unserer Erfahrung eine völlige Fehleinschätzung. Tatsache ist, dass Programmierer, die über die Funktionsweise des Gesamtsystems genau Bescheid wissen, viel effizienter arbeiten - auch und insbesondere weil sie Problembereiche besser erahnen können, als wenn sie einfach gegen Blackboxmodule arbeiten. Darüber hinaus ist es erstaunlich, wie viel das menschliche Gehirn an Informationen aufnehmen und auch sinnvoll verarbeiten kann. Es ist jedem beteiligten Programmierer zuzutrauen (und darum auch die Zeit dafür vorzusehen), dass er sich mit der gesamten vorliegenden Dokumentation vertraut macht - oft führt das wie in den schon oben genannten Fällen dazu, dass einem Programmierer Unstimmigkeiten oder Schwächen im System auffallen, die bisher noch keiner der Designer gesehen hat.

Tipps zur Anwendung:

In einer *Technischen Dokumentation* könnten zum Beispiel folgende Informationen zu finden sein:

- welche Fehlerzustände können auftreten
- wo werden Fehler in welcher Form geloggt
- wie ist die Software zu konfigurieren
- was haben die diversen Konfigurationseinstellungen für Auswirkungen auf die Funktionsweise der Software
- wo sind potenzielle Problembereiche
- wo sind Fehler oder schlechte Performance zu erwarten und wie kann man manuell in diesen Fällen nachhelfen
- warum wurden gewisse Implementierungsentscheidungen getroffen (das kann sich als besonders wichtige Information herausstellen, wenn es darum geht, das System zu warten und manche dieser Entscheidungen im Lichte der Erfahrung abzuändern)
- welche Software von Drittherstellern wurde wo und warum verwendet
- nach welchen Prinzipien arbeitet die Software, wenn sie ein gewisses Ziel verfolgt
- wie wurden die Daten modelliert
- wie wird auf Datenbanksysteme zugegriffen (im Hinblick etwa auf Normalformen bei relationalen Datenbanken, die ja oft aus Performancegründen nicht eingehalten werden können)
- etc.

Die *Technische Dokumentation* kann also sehr umfangreich werden - es gilt, einen Kompromiss zwischen möglichst vollständiger Beschreibung und Größe zu finden - sie sollte auf jeden Fall nicht so groß werden, dass potentielle Leser davon abgeschreckt werden, sie zu lesen. Im Falle, dass die *Technische Dokumentation* zu groß wird, muss man die Beschreibung in mehreren Abschnitten mit unterschiedlichem Grad von Detaillierung erstellen, sodass der Leser einen "drill down" Ansatz verfolgen kann.

5. Design Dokumentation

Design ist die wohl wichtigste Phase in der Softwareentwicklung in dem Sinne, dass hier am meisten schiefgehen kann, was später schwer zu korrigieren ist. Hier erfolgt sozusagen die "Technifizierung" der fachlichen Anforderungen, d.h. der Analyseergebnisse. An dieser Stelle wird versucht, zum ersten Mal die Grätsche zwischen Kundenwunsch und technischer Machbarkeit zu machen. Daraus allein folgt schon, dass es extrem wichtig ist, alle hier getroffenen Entscheidungen festzuhalten – typischerweise führen diese Entscheidungen ja dazu, dass gewisse Einschränkungen in der Funktionalität gemacht werden. Ein funktionsfähiges Softwaresystem ist keine "eierlegende Wollmilchsau". Je spezifischer seine Fähigkeit ist und je mehr sie sich einschränken lässt, desto höher sind die Chancen darauf, ein brauchbares, robustes System zu entwickeln. Man kann wohl sagen, dass es für jede der getroffenen Einschränkungen zumindest eine Person gibt, die sie eines Tages in Frage stellen wird - das ist dann genau der Zeitpunkt, wo man froh ist, den Sinn der Einschränkung bzw. ihre Existenz dokumentiert zu haben.

Eine wichtige Aufgabe der Design-Phase ist, die Modellierung der Realweltkonzepte unter Einbeziehung von technischen Machbarkeitsaspekten zu dokumentieren. D.h. also etwa, die Dokumentation des Datenbankdesigns (Tabellen, Spalten, deren Typen, Relationen, etc.). Es kann nicht genug betont werden, wie wichtig ein gut dokumentiertes Datenbankdesign ist. Da es ja immer wieder nötig wird, Spalten oder Tabellen bzw. Relationen hinzuzufügen oder zu ändern, wenn sich die zu verarbeitenden Daten ändern, ist es extrem wichtig, allen Beteiligten eine klare Vorstellung über die Ideen des vorliegenden Datenbankdesigns zu vermitteln, damit jede Erweiterung ins Konzept passt. Dazu gehören Namenskonventionen ebenso wie detaillierte Beschreibungen der fachlichen Bedeutung von Tabellen und deren Spalten - man beachte, dass eine solche fachliche Beschreibung aus einem Entity-Relationship Diagramm allein nicht hervorgeht!

Tipps zur Anwendung:

In die *Design Dokumentation* gehören auch alle anderen Formen von Dokumentation, welche die Teile in ihrem Zusammenspiel abbilden, wie etwa UML Diagramme, welche die Relationen zwischen den verschiedenen beteiligten Klassen aufzeigen. Diese Form der Dokumentation ist erfahrungsgemäß am anfälligsten dafür, dass sie nicht mehr mit den realen Gegebenheiten übereinstimmt, da sich das System weiterentwickelt hat, die Dokumentation jedoch nicht nachgezogen wurde. In diesem Fall gilt jedoch wiederum: besser ein Diagramm, das die Wahrheit nicht mehr ganz wiedergibt, als gar keines. Gewisse Grundideen bleiben auch erhalten, wenn sich ein System entwickelt und an einem Relations Diagramm können solche Ideen am schnellsten erkannt werden. Es ist nicht zu empfehlen, bei Systemen, die wahrscheinlich eine relativ starke Evolution vor sich haben, bei solchen Diagrammen zu sehr ins Detail zu gehen - die genaue Anzahl und der genaue Typ etwa von Parametern einer Funktion oder Methode haben dann eine sehr geringe "Halbwertszeit"; es wird dann kaum jemand die Zeit erübrigen können, die Dokumentation nachzuführen. Wiederum gilt oben gesagtes: das geschriebene Wort zur Bedeutung der beteiligten Klassen kann durch kein noch so ausgefeiltes Diagramm ersetzt werden - das Diagramm gibt die Übersicht, der Fliesstext erläutert das Detail.

6. Reihenfolge in der Erstellung

Die hier verwendete Abfolge entspricht nicht der Reihenfolge, in der ein *neues* Softwareprojekt dokumentiert werden sollte. In diesem Fall wird man als erstes die Designdokumente entwickeln, dann eine technische Dokumentation samt Schnittstellenbeschreibung erstellen und zuletzt, während des eigentlichen Codierprozesses, die Inline Kommentare vorsehen.

Warum wurde hier dennoch diese umgekehrte Reihenfolge gewählt?

Es ist eine traurige aber wohlbekannte Tatsache, dass die meisten der heute existierenden Softwarelösungen, insbesondere jene für den vertikalen Markt (d.h. also maßgeschneiderte Lösungen die in meist relativ kurzer Zeit für wenige Kunden entwickelt werden - etwa Applikationen für Versicherungen und Banken), wenig bis gar keine Dokumentation aufweisen. Wenn man nun anfängt, Dokumentation in solche Projekte einzuarbeiten, so sollte man in der hier verwendeten Abfolge vorgehen: die vitalste Form von Dokumentation, die auf keinen Fall fehlen sollte, ist die *Inline Source Dokumentation*, da sie entscheidend zur Wartbarkeit eines Programms beiträgt. Die zweitwichtigste Form ist die detaillierte *Schnittstellenbeschreibung* - wiederum ist die Wartbarkeit der Software davon betroffen; darüber hinaus können bei dieser Form der Dokumentation (wie auch bei technischer und Designdokumentation) Unstimmigkeiten aufgedeckt werden, welche die Software unnötig verkomplizieren. *Technische Dokumentation* sollte ebenfalls erstellt bzw. gewartet werden, um stets einen Überblick über das Zusammenspiel der Teile der Software zu haben. *Designdokumente* sind im Falle eines bereits fertigen Softwaresystems natürlich am entbehrlichsten - wie jedoch schon erwähnt, erfüllt aber auch die Dokumentation eines Datenbankdesigns einen besonderen Zweck für die Programmwartung.

7. Resümee

Tatsächlich verzichtbar ist wohl keine der genannten Dokumentationen. Die Zeit, die man dafür investieren muss, rentiert sich in jedem Falle - sie kommt als Zeitersparnis bei Fortentwicklung und Wartung des Systems (dies teilweise durch weniger neu eingeführte Fehler und teilweise durch kürzere Entwicklungszeiten) auf jeden Fall vielfach zurück.

Es kann hier nicht genug davor gewarnt werden, als einzige Dokumentationsform "Mundpropaganda" einzusetzen. Was hier selbstverständlich klingt, wird dennoch von wenigen Softwarefirmen tatsächlich beherzigt; darum sind hier nochmals die wichtigsten Argumente für professionelle Dokumentation aufgezählt:

- Im Falle fehlender Dokumentation steht und fällt jedes Projekt in viel stärkerem Maß als nötig mit dem Wissen einzelner Personen. Im Besonderen, wenn eine gewisse Personalfluktuation erwartet wird, darf man sich also in keinem Fall darauf verlassen, dass Schlüsselpersonen immer zur Verfügung stehen werden.

- Es ist für die beteiligten Programmierer, Designer und Administratoren oft mühsam und unangenehm, alle anderen "abklappern" zu müssen um ein Stück triviale Information zu bekommen. In Folge wird darauf verzichtet, die exakte Information einzuholen und stattdessen von ad hoc Plausibilitätsannahmen ausgegangen, die oft falsch sind. Die Konsequenz ist wieder: unnötige Fehler in der Software.

- Noch einmal muss der Wert des Übersichtsbildes betont werden. Jeder der beteiligten Programmierer bzw. Designer sollte sich ein geistiges Bild vom Gesamtsystem machen können - das ist natürlich kaum möglich, wenn das Wissen in kleinen Brocken auf viele Leute verteilt ist.

Darum: Dokumentiere früh und dokumentiere oft; ein viel übersehenes Faktum in diesem Zusammenhang ist auch die Tatsache, dass es durchaus Spaß macht, Dokumentation zu schreiben - daraus kann auch ein nicht unerheblicher Motivationsschub für die beteiligten Programmierer und Designer folgen.

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.

Harald Nowak
Wien, im Dezember 2002

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.