1 Das Ziel: Vertrauen in das Projekt

1.1 Vertrauen des Kunden

Was erwartet der Kunde ?

- Projekt erfüllt die Anforderungen
- Keine versteckten Mängel
- Keine Kostenfalle durch nie endende Wartung
- Keine Abhängigkeit vom Entwicklerteam

Was bieten wir dem Kunden an ?

- Kunde hat jederzeit Kontrolle über das Projekt
  - Wir können jederzeit ein lauffähiges Programm ausliefern.
  - Kurze Iterationszyklen (3 bis 6 Wochen bis zum nächsten Release, wobei jedes Release einen Mehrwert bietet) erlauben dem Kunden aus dem Projekt auszusteigen, bzw. nachzuverhandeln; und garantieren uns schnelles Feedback.
  - Das Projekt wird immer in einem Zustand gehalten, indem auch ein anderes Team die Arbeit übernehmen kann.
  - Wir garantieren leichte Einarbeitung neuer Mitarbeiter. Fluktuation von Mitarbeiter bringt das Projekt nicht zum Scheitern.
  - Zeitverzögerungen werden rechtzeitig erkannt.
- Wir garantieren hohe Qualität des Produkts
  - Wir achten nicht nur auf Korrektheit sondern auch auf Testbarkeit.
  - Durch automatisierte Tests ist nachkontrollierbar was getestet worden ist.
  - Die Tests sind jederzeit wiederholbar.
  - Tests und Codeentwicklung laufen nicht auseinander, da Tests in SourceCode integriert ist (im Gegensatz zu Testdrehbücher).
  - Wir garantieren Testabdeckung des Sourcecodes von über 80%
  - Kunde hat Zugriff auf den Source Code und kann die Qualität durch eigene IT-Leute bzw. Analyse Programme testen.
  - Keine hohen Wartungskosten aufgrund schlampiger Entwicklung
- Projekt genügt den Anforderungen
  - Anforderungen werden so formuliert, daß sie vom Anwender verstehbar und testbar sind.
  - Verwendung von Oberflächentests simulieren Aktionen des Anwenders.

1.2 Vertrauen des Projektleiters

- Truck Faktor wird minimiert, keine Abhängigkeit von Wissen einzelner Mitarbeiter
- Testbar was abgeschlossen ist, und es bleibt i.a. abgeschlossen
- Verschieben von Mitarbeitern von schlecht ausgelasteten Projekten in Projekten mit viel Arbeit leichter möglich
- Projektzustand ist messbar. 80% zu 20% Faktor fällt weg.
- Anfänger leichter einzubinden, da hoher Lernfaktor.
- Keine Abhängigkeit von teuren Tools. Investition in Mitarbeiter

1.3 Vertrauen der Entwickler

- i.a keine Probleme Urlaub zu bekommen.
- gleichmaessigere Auslastung (weniger Stress)
- Vertrauen das der Code auf dem man aufbaut funktioniert
- Nicht das Gefühl allein gelassen zu sein.
- Profitiert von Wissen seiner Kollegen
- Hoher Lernfaktor ==> Schnelle Aneignung von IT-Allgemeinwissen.
- Keine Angst Aenderungen durchzuführen.

2 Der Weg: Effektives Projektmanagement

2.1 Grundprinzip Fehlerfreiheit

Das Ziel ist es zu jedem Zeitpunkt dem Kunden ein fehlerfreies Programm übergeben zu können. Deshalb lautet die Grundprinzipien:

Keine Weiterentwicklung solange noch Fehler enthalten sind
Was nicht explizit getestet wurde, gilt als nicht korrekt

Fehlerfreiheit bedeudet hier, dass die innere Qualität der Software sehr hoch ist.

- alle Anforderungen die implementiert sind, entsprechen den Erwartungen des Kunden; allerdings muessen nicht zu jedem Zeitpunkt alle Anforderungen implementiert sein. (sonst wären wir ja fertig).
- Ist ein Programmteil in einem unsauberen Zustand, und man kann ihn nicht schnell beheben, möchte (oder muss) aber weiterentwickeln; so wird der unsaubere Programmteil weggeschmissen, und durch einen Dialog 'Noch nicht implementiert' ersetzt.
  - Der Kunde bekommt nicht den falschen Eindruck, daß der Programmteil schon gelöst ist.
  - Die Entwickler haben nicht die Chance auf fehlerhaften oder unsauberen Code aufzubauen.
- Die Akzeptanz Tests muessen erst am Ende des Projektes vollständig korrekt durchlaufen.
- Die UnitTests müssen zu jedem Zeitpunkt 100% erfolgreich durchlaufen.
- Jede Funktionalitaet wird durch mindestens einen UnitTest abgedeckt. Es gibt keinen Code, der nicht durch UnitTests abgedeckt ist. (Ausnahmen siehe Kapitel über UnitTests).
- Alle Programmierstandards werden eingehalten.
- Es gibt keinen redundanten Code.

Unbedingt zu verhindern ist es, daß man viele offenen Punkte mit sich schleppt, die immer nur fast fertig sind

2.1.1 Die Broken Window Theorie

Nach der Broken Window Theorie zieht Schmutz und Zerstörung weitere Verschmutzungen und Zerstörungen an. In einem Versuch in den USA wurde ein Auto mit geöffneter Motorhaube in New York abgestellt. Innerhalb weniger Minuten wurde es geplündert. Ein vergleichbares Auto blieb abgestellt in Palo Alto eine Woche unangerührt. Dann hat man gezielt eine Scheibe des Autos zerstört. Dies kaputte Scheibe zog weitere Beschädigungen nach sich und in kurzer Zeit

wurden die anderen Scheiben auch eingeschlagen, die Reifen wurden geklaut etc. In New York werden daher Graffitis sofort entfernt, wenn sie entdeckt werden. Man geht davon aus, dass es Sprayern einfacher fällt, eine Wand mit einem Graffiti zu verschönern, wenn sie nicht die Ersten sind.

(aus dem Buch: Software entwickeln mit XP - Erfahrungen aus der Praxis, Autoren: Martin Lippert, Stefan Roock, Henning Wolf)

Wer die Qualität eines Projekts nicht misst, kann nie aussagekräftige Prognosen über den Projektfortschritt tätigen.

In den meisten Projekten verlangt der Projektleiter lediglich Schätzungen der Entwickler wie weit ihre Tätigkeiten fortgeschritten ist. Die Entwickler schätzen meist sehr positiv, da in vielen Unternehmen Zaudern als Schwäche angesehen wird. Ist der Code auf den die Entwickler aufbauen fehlerhaft, haben die Schätzungen keine Chance realistisch zu sein. Viele Entwickler wagen es (zurecht) nicht den existierenden Code zu verbessern, da auch andere Entwickler auf diesen Code aufbauen, und eine Änderungen bei diesen zu Fehlern führt. Statt existierenden Code zu verwenden, implementieren diese Entwickler dann die Funktionalität erneut. So entstehen viele Versionen derselben Funktionalität, jede Version hat hat ihre eigenen Schwächen und ist ausserhalb des eigenen Kontextes schwer verwendbar. Mit fortschreitender Projektdauer hat keiner mehr den Überblick, der Projektzustand lässt sich nicht mehr messen. Änderungen an Anforderungen bewirkten Änderungen an vielen verschiedenen Teilen des Codes. Da die Codeteile voneineinander abhängig und ineinander verwoben sind, ziehen kleine Änderungen grosse Nachbesserungen auf sich.

In einem auswärtigen Projekt an dem ich in der Schlussphase mitgearbeitet haben, hatten die Entwickler einem Monat vor Abgabetermin um eine Aufschiebung um einen Monat gebeten. Als dieser Monat zu Ende war benötigten die Entwickler noch 2 weitere Monate. Danach verdoppelte sich jeweils diese Zeitspanne noch 2 bis 3 mal, bis man das Projekt dann entgültig als gescheitert erklärte. Die notwendige Funktionalität war zwar schon lange implementiert, aber trotz sehr guter Entwickler schaffte man es nicht die Software zu stabilisieren. Das Problem lag in erster Linie im Management. Auf innere Qualität wurde bewusst nicht geachtet. Um schnell Erfolge nachweisen, und den ursprünglich sehr engen Zeitplan einhalten zu können, wurden die Entwickler angehalten statt den existierenden Code zu stabilisieren, möglichst schnell neue Funktionalität hinzuzufügen. Die eigentliche Kunst von Projektmanagement liegt nicht im Erstellen von Gantt-Diagrammen, sondern im Messen des Projektzustands, sowie die Kommunikation zwischen den Projektbeteiligten zu optimieren. Nur eine hohe innere Qualität der Software garantiert, dass die Komplexität von Software nicht mit neuen Anforderungen exponential ansteigt.

2.2 Anforderungsmanagement

2.2.1 Zentrale Bedeutung von Anforderungsmanagement

Ein Erfahrungsbericht

Ich hatte vom meinen Projektleiter als Aufgabe bekommen die Bauraumsuche zu lösen. Die Anforderung schien eindeutig und klar gestellt. Wenn ein neues Bauteile in einen Motor eingebaut wird, so soll kontrolliert werden ob dafür überhauptnoch Platz ist. Die Bauteile werden vereinfacht als Quader mit x,y und z Achse repräsentiert.

Zunächst hatte ich mich gewundert, da ich dies am Anfang für ein triviales Problem hielt. Ich dachte ich muss nur die Ecken des neuen Bauteils kontrollieren, ob diese im Rauminhalt eines anderen Bauteils befindet. Mein Projektleiter verdeutlichte mir mit 2 Büchern daß dies nicht der Fall ist. Ein Buch könnte ein anderes durchstossen.Es reicht nicht die Ecken zu kontrollieren, eigentlich müsste ich jeden Punkt des Rauminhaltes in einem Buch kontrollieren.

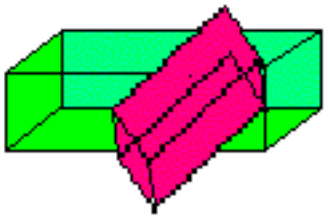

Abb. 2.1: Bauraumsuche

Das Problem war mit Abiturmathematik nicht zu lösen. Ein promovierter Mathematiker in unserem Haus verbrachte einige Stunden mit diesem Problem, und verwies mich auf eine C++ Lösung von einer amerikanischen Universität. Ich selbst hatte grosse Schwierigkeiten die Lösung zu verstehen. Ich erinnerte mich aber an die lineare Optimierung. Eine Methode die ich im Studium im Nebenfach Energietechnik erlernt hatte, um die Wirtschaftlichkeit von Kraftwerken zu berechnen.

Die Lineare Optimierung ist nicht unbedingt performant, und liefert unter gewissen Bedingungen gar keine Lösungen.Deshalb rief ich (leider viel zu spät) beim Kunden den fachlichen Ansprechpartner an. Das Gespräch dauerte keine Minute. Die Aussage war: Wir bauen keine Teile schräg in den Motor, sondern immer nur senkrecht ein.

Damit war aus einem komplexen dreidimensionalen Problem, ein triviales zweidimensionales geworden. Ich musste nur überprüfen ob sich 2 Rechtecke schneiden.Ich möchte die Kosten die durch Kommunikationsproblemen mit dem Kunden entstehen, lieber nicht abschätzen.

Probleme ergeben sich meistens nicht aus den Fakten die unbekannt sind, sondern aus denjenigen die man fälschlicherweise glaubt zu wissen.

Primäre Aufgaben einer Anforderung

- Kommunikationsgrundlage
- Vertragsgrundlage
- Grundlage für Integration, Wartung
- Grundlage für Systemarchitektur

Die sieben Probleme der Systemanalyse

- Unklare Zielvorstellung für das System
- Hohe Komplexität der zu lösenden Aufgaben
- Kommunikationsprobleme - Sprachbarrieren zwischen den Projektbeteiligten
- sich ständig veränderte Ziele und Anforderungen
- schlechte Qualität der Anforderungen

- Goldrandlösungen
- ungenaue Planung und Verfolgung des Projektes basierend auf ungenaue Anforderungen.

2.2.2 Klassifikation von Anforderungen

Man kann Anforderungen grob in 3 Bereiche einteilen.

Funktionale Anforderungen

Eine funktionale Anwendung ist eine Leistung die das Produkt erbringen muss.

Beispiele:

Das System ermittelt täglich alle Kunden die im letzten Jahr Produkte fuer insgesamt weniger als 100 Euro bestellt haben und aktuell Geburtstag haben. Diesen Kunden wird die Standard Geburtstags E-Mail zugesendet

Das System ermittelt täglich alle Kunden die im letzten Jahr Produkte fuer mehr als 100 Euro bestellt haben und in 3 Tagen Geburtstag haben. Für diese Kunden wird ein Auftrag generiert, die Ihnen das Standard Werbegeschenk des aktuellen Monats zusendet

Nichtfunktionale Anforderungen

Eine nichtfunktionale Anforderung ist ein Qualitätsmerkmal das ein Produkt hat.

Beispiele:

Die Zeitspanne von der Eingabe der Kundennummer, bis die Auftraege des Kunden aufgelistet werden, darf maximal 5 Sekunden betragen

Jedes Fenster, in dem Daten eingegeben werden, hat eine Symbolleiste. Mit dem ersten Icon kann man das Fenster schliessen; dieses Icon ist immer ganz links in der Symbolleiste. Mit dem letzten Icon ruft man das Handbuch auf. Das letzte Icion ist immer ganz rechts in der Symbolleiste

Constraints

Dieses sind Einschrängen und Randbedingungen, die eingehalten werden sollen.

Beispiel:

Das Produkt muss sowohl unter Windows XP und Unix laufen

2.2.3 Qualitätskriterien von Anforderungen

- Jede Anforderung wird einzeln getestet. Beispiele fuer Qualitätsktriterien sind:
  - Ist die Anforderung wirklich eine Anforderung, oder schon eine mögliche Lösung zu einer Anforderung
  - Ist die Anforderung sowohl fuer Kunde und Entwickler verständlich.
  - Ist die Einhaltung der Anforderung testbar
  - Ist der Kontext in dem die Anforderung steht beschrieben
    - zu welchem UseCase gehört die Anforderung?
    - Verweis auf externe Dokumente vorhanden?

    - Von wem kommt die Anforderung ?
    - Angabe der Priorität vorhanden ?
    - Rechtliche Relevanz der Anforderung ?
- Die Anforderungen werden in der Gesamtheit getestet.
  - Widersprechen sich zwei oder mehrere Anforderungen
  - Fehlen noch wichtige Anforderungen, sind alle wichtigen UseCases gefunden.

Aus den Anforderungen müssen Akzeptanzkriterien abgeleitet werden können, die den (Nicht) Erfolg des Projekts messen

Das Auffinden der Anforderungen ist i.a. ein sehr komplexer iterativer und inkrementeller Prozess. Dieser Prozess erstreckt sich waehrend der gesamten Lebenszyklus des Projekts. Anforderungen ändern sich, es kommen neue hinzu, alte Anforderungen werden gelöscht.

In der Literatur sind verschiedenartige Prozesse aufgelistet. Ich versuche die Gemeinsamkeiten herauszufinden:

- Beschreibung des Projektziels
- Finden der Stakeholders. Stakeholders sind Personen, die Interesse an diesem Projekt haben (Anwender sind nur ein Teil davon).
- Beschreibung der Systemgrenzen.
  - Abgrenzung gegenüber anderen Systemen.
  - Welche Systeme sind an dem zu erstellenden System angeschlossen.
  - Welche Datenströme kommen herein, und welche kommen heraus, und in welcher Form sind diese Ströme
- Wann und wie wird das zu erstellende System getriggert.
- Finden der UseCases bzw. Geschaeftsprozesse.
- Finden der funktionellen und nicht funktionellen Anforderungen.
- Festlegen der Rahmenbedingungen
  - z.B.: Was darfs kosten.
  - z.B: Wann ist Abgabetermin
- Qualitätskontrolle
  - jeder einzelnen Anforderung.
  - Gesamtheit aller Anforderungen.

2.2.4 Verwaltung von Anforderungen

In leichtgewichtigen Methoden wie z.B Extreme Programming, werden Anforderungen auf Karteikärtchen beschrieben. Dies fuktioniert meines Erachtens nur, wenn man den Kunden direkt in das Projekt integriert, das notwendige Wissen ueber das ganze Team verteilt ist, sowie sofortiges FeedBack vom Kunden garantiert ist. Schwergewechtige Methoden (wie z.B Rational Unified Process) benötigen meines Erachtens eine Anforderungsdatenbank. Anforderungen sollte man nach verschiedenen Kriterien sortieren und filtern koennen.Typische Kriterien sind:

- rechtliche Relevanz
- Use Case bezogen
- Typisierung (z.B Look&Feel Anforderung oder Sicherheitsrelevante Anforderung)
- Fenster bezogen

Neben Sortierung und Filterung von Anforderungen sind Versionierungen notwendig. Änderungen

von Anforderungen sind Grundlage für Mehraufwendungen; und damit für Vertragsnachbesserung.

Eine zentrale Bedeutung nehmen die Stakeholders ein. Wer ist der Ansprechpartner fuer eine Anforderung?. Wer muss benachrichtigt werden; wenn sich eine Anforderung ändert?.

Aus den Anforderungen in den Datenbanken laesst sich ein Word bzw. HTML Dateien erzeugen, und somit auch den Kunden Zugriff auf die Anforderungen zu ermöglichen. Ggf. koennen bei Änderungen der Anforderungen automatisch die betroffenen Stakeholders informiert werden.

2.3 Akzeptanz- und UnitTests

2.3.1 Unterschiede zwischen Akzeptanz und Unittests

Akzeptanztests testen das gesamte System aus Sicht des Kunden von aussen(BlackBox Test). I.a. simulieren sie ein Testskript das ein Testteam vor Inbetriebnahme einer neuen Version durchspielt. Die Akzeptanztests werden in enger Zusammenarbeit mit den Kunden geschrieben.

Unittests testet die Einzelteile (i.a auf Klassenebene)des Systems aus Sicht des Programmierers (WhiteBox Test). Es wird i.a. jeweils nur eine Klasse getestet, ggf. noch die Interaktion mit angrenzenden Klassen. Es sollte keine Funktionalität in einer Klasse geben, die nicht durch Unittests abgedeckt wird.

Was wird getestet ?

- Akzeptanztest testen das gesamte System, und können von Anwender nachvollzogen werden. Akzeptanztests werden in enger Zusammenarbeit mit dem Kunden definiert.
- Unittests testen eine Programmiereinheit i.a. eine Klasse. (Dies ist nicht zwingend). Unittests werden von Programmierer entwickeln.

Wie häufig wird getestet ?

- Akzeptanztests werden täglich oder wöchentlich ausgeführt. Nach jeder Integration und vor einer Auslieferung
- Unittests werden eigentlich andauernd, nach jeder Änderung ausgeführt. Also in 2 bis 15 Minuten Takt. Es werden dabei immer alle Unittests ausgeführt und nicht nur dijenigen an den entsprechenden Klassen gearbeitet worden ist.

Was sind die Erfolgskriterien für Tests ?

- Akzeptanztests dürfen ruhig fehlschlagen. Der Erfolg sollte allerdings monoten wachsend sein. Erst am Projektende sollten alle Akzeptanztests korrekt ablaufen.
- Unittests sollten jederzeit immer zu 100 % korrekt ablaufen. Sollte einer schief gehen, wird nicht mehr weiterentwickelt sondern der Fehler behoben.

Testdauer ?

- Da Akzeptanztests nicht allzuhäufig ausgeführt werden, können diese Tests ruhig einige Stunden andauern.
- Unittests müssen sehr schnell ablaufen, da diese im Minutentakt ausgeführt werden sollen.

Wie testet man Zugriff auf externe Systeme ?

- Akzeptanztests greifen i.a. auf externe Systeme zu. Allerdings kann nur getestet werden, wenn das Verhalten der externen Systeme konstant ist. Da z.B. bei Unterschiedlichen Datenbankinhalt die Ausgabe verschieden ist, muss vor der dem Test. ein definierter konstanter Datenbankbestand eingeladen werden. Ansonsten sind die Akzeptanztests nicht wiederholbar.
- Unittests greifen nie auf externe Systeme zu, diese werden mit Mock Objekten simuliert.

Mit welcher Sprache testet man?

- Akzeptanztests werden oft mit einer Skriptsprache geschrieben. Am besten ist es wenn die Anwender sie nachvollziehen können. Typisch sind Testsysteme die die Aktionen eines Benutzers an der Oberfläche aufzeichnen.
- Unittests schreibt man i.a. immer in der Sprache in der man entwickelt.

2.3.2 Vorgehensweise Akzeptanztests

Fuer Akzeptanztests braucht man eine spezielle Testdatenbank mit konstant definierten Inhalt. Dieser Inhalt wird einmal gesichert, damit man vor Anstossen der Akzeptanztests diese wieder einladen kann.

Ein Akzeptanztest simuliert i.a. eine oder mehrere Benutzeraktionen des zu testeten Programms. Diese Benutzeraktionen werden durch Events programmatisch ausgeloest. Aufgrund der Benutzeraktionen erfolgen mehr oder weniger komplexe Berechnungen.

Um die Korrektheit der Ergebnisse zu ueberpruefen werden Zustand von Objekten im Image bzw. Eintraege in den Datenbanken ueberprueft.

Ggf. muss am Ende eines Tests die Testdatenbank wieder bereinigt werden. Ein Testframework protokolliert fuer jeden einzelnen Test den Erfolg. Ggf. verbindet man einzelne Tests mit Performance Messungen.

Zur Not kann auch JUnit fuer Akzeptanztests verwendet werden. Im Gegensatz zu Unittests sind bei Akzeptanztests fehlgeschlagene Tests kein Beinbruch. Erst am Projektende sollten auch alle Akzeptanztests laufen. Der Nachteil von JUnit für Akzeptanztests ist daß der Kunde damit nicht umgehen kann.

Am besten ist es wenn der Kunde Akzeptanztests selbst aufrufen und die Ergebnisse überprüfen kann. Das Open-Source Tool Fitnesse ist ein erweiterter Wiki-Server. Er dient gleichzeitig zur Dokumentation und zum Ausführen von Java-Tests. Mit Fitnesse kann die Dokumentation einer Anforderung und die dazugehörigen ausführbaren Akzeptanztests nebeneinander innerhalb einer einzigen HTML Seite definiert sein.

2.3.3 Vorgehensweise Unittests

Folgende Vorgehensweisen werden empfohlen:

1. Lasse zuerst alle Tests laufen, um sicher zu gehen, dass man nicht auf fehlerhaften Code aufbaut. Korrigiere ggf. erst den Fehler.
2. Schreibe immer zuerst ein Test, dann den dazugehörigen Code
3. Implementiere immer genau soviel, dass die Tests korrekt ablaufen.

4. Entwickle in möglichst kleinen Schritten
5. Geht ein Test schief, wird nicht weiterentwickelt sondern erst der Fehler korrigiert
6. Refaktoriere nach jedem Test, falls möglich.
7. Oft macht es Sinn, ersteinmal absichtlich im Code einen Fehler einzubauen, um zu testen ob der TestCase korrekt implementiert ist.

Beispiel:

Zu Entwickeln ist eine Klasse Mitarbeiter

1. Lasse alle vorherigen Tests laufen, diese muessen zu 100% laufen.
2. Zunächst soll das Erzeugen eines Mitarbeiters getestet werden.
3. Versioniere den Code, um ggf. falls alles schiefgeht schnell zu einem korrekten Ausgangszustand zurückkehren kann.
    1. Überlege wie die einfachst möglichste Schnittstelle aussieht um einen Mitarbeiter zu erzeugen
    2. Schreibe einen TestCase, mit dieser Schnittstelle, der überprüft ob das Erzeugen eines Mitarbeiters funkioniert.
    3. Lasse die TestCases laufen. Es wird ein Compiler Fehler geben, da die Schnittstelle noch nicht implementiert ist.
    4. Implementiere die Schnittstelle gerade soweit, dass der Test ohne Compilerfehler abläuft.
    5. Lasse die TestCases laufen.Der aktuelle TestCase wird zwar ablaufen aber einen Fehler anzeigen,da die Funktionaliät
    6. ja immer noch nicht implementiert ist. Wird kein Fehler gemeldet, so ist der Test falsch.
    7. Implementiere die Funktionalitaet, ohne auf Refaktorierungen zu achten.
    8. Lasse **alle** TestCases laufen. Korrigiere den Code solange bis alle TestCases korrekt ablaufen.
    9. Kann man die Tests oder den SourceCode refaktorieren?. Falls ja, muss dies gemacht werden. Ggf. vorher den Code wieder versionieren.
    10. Lasse nach der Refaktorisierung wieder alle TestCases durchlaufen, um die Korrektheit der Aenderungen zu garantieren.
4. Nun soll garantiert werden, dass es nicht moeglich ist zwei Mitarbeiter zu erzeugen, die dieselbe Personalnummer haben.
    1. Schreibe zunaechst einen TestCase der die Funktionalitaet überprueft
    2. Nun muss wieder beim Ablauf aller TestCases ein Fehler auftreten.
    3. Implementiere die Funkionaliaet gerade soweit dass die TestCases laufen.
    4. Kontrolliere durch den Ablauf aller TestCases, dass die Funktionalität korrekt ist.
    5. Refaktoriere
    6. Es werden wieder alle TestCases durchlaufen, um zu testen daß durch das Refaktorieren keine Fehler entstanden sind
5. Für alle anderen Funktionaliäten gehen wir genauso vor:
    1. Erst einen Test schreiben
    2. Tests laufen lassen, es muss einen Fehler ergeben.
    3. Implementierte gerade soweit, daß der Test korrekt abläuft
    4. Tests laufen lassen, falls Test nicht korrekt muss ein Fehler im Test oder im Code sein. Fehler muessen sofort behoben werden.
    5. Refaktoriere
    6. Alle Tests laufen lassen, um die Korrektheit des Refaktorierens zu garantieren.

Achtung:

1. Wenn alle Tests korrekt durchlaufen, heisst das noch lange nicht daß der Code fehlerfrei

ist. Ein korrekter Durchlauf sagt nichts über die Qualität der Tests aus.
2. Das Entwickeln von Unittests ist alles andere als trivial und muss gelernt werden. Oft kommt die Projektleitung erst auf die Idee Unittests einzuführen, wenn die Qualität des Codes schon sehr schlecht ist. Unittests im Nachhinein einzuführen ist aber auch für erfahrenen Unittester eine schwere Herausforderung. Wenn die Qualität des existierenden Codes sehr schlecht ist, hilft nur noch neu schreiben.

2.3.4 Schwierigkeiten Unittests

Das Ziel ist es mit Unittests eine 100% Testabdeckung zu erreichen. Dies ist allerdings i.a. nicht moeglich.

- **Schwierigkeiten Benutzungsoberflaeche zu testen**

Die einfachste Lösung ist Benutzungsoberflaechen nicht zu testen. Dies geht nur wenn die Benutzungoberflächenobjekte sehr duenn sind. D.h es werden nur die Widgets dargestellt, jegliche Logik wird ausgelagert. Man trennt die Benutzungsoberflaeche in eine Interaktionsschicht und eine Funktionsschicht. (siehe OO-Design Kurs). Die Objekte der Funktionsschicht wird mit Unittests überprueft.

- **Zugriff auf externe Systeme sind teuer**

Zugriff auf Datenbanken und Hostsysteme dauern lange, und man kann nichts testen falls diese ausfallen. Die Idee ist diese Systeme zu simulieren(Mockobjekte). Es gibt im Internet existierende Frameworks für Mockobjekte. Falls man Mockobjekte falsch programmiert, beteht die Gefahr dass sie die Tests verfälschen. Der richtige Umgang mit Mockobjekten kann aber Tests erheblich vereinfachen.

- **Threads**

Threads werden verwendet um Prozesse im Hintergrund ablaufen zu lassen. So kann ein Anwender Eingaben in seinen Fenstern betätigen, während der Rechner komplexe Algorithmen berechnet. Mit Threads haben wir leider oft ein nichtdeterministisches Verhalten. Je nachdem welcher Thread schneller oder langsamer abläuft, kann das Gesamtverhalten korrekt oder falsch sein.

Eine mögliche aber unschöne Lösung dieselben Unittests öfters als einmal ablaufen zu lassen. Die Wahrscheinlichkeit einen Defekt zu finden wird höher, aber den Beweis auf Korrektheit haben wir damit nicht.

Zunächst muss überprüft werden, ob die Threads überhaupt benötigt werden. In vielen Fällen bekommen wir durch Einführung von Threads einen Performance Verlust. Oft kann man statt einem Thread (d.h. leichtgewichtiger Prozess der sich mit anderen Prozessen gemeinsamen Adressraum teilt) einen richtigen Prozess verwenden (ein eigenstaendiges Programm, das seinen eigenen Adressraum hat). Wenn man nicht schon auf Threads verzichten kann, sollte man sich an die gängigen Patterns (siehe Buch von Frank Buschmann) halten.

- **keine Moeglichkeit auf private Methoden zuzugreifen**

Das Problem gibt es in Smalltalk zum Glück nicht, da sind alle Methoden öffentlich. In meinen ersten Erfahrung in Java löse ich das Problem dadurch, dass ich kaum private Methoden verwende (auch keine optimale Loesung). Da ich die Tests im selben Package habe wie die zu

testende Klassen kann ich auch protected bzw default Methoden testen.

Laut Kent Beck und Frank Westphal sollten private Methoden nicht getestet werden, da sie indirekt von öffentlichen Methoden aufgerufen werden. Sie meinen falls wirklich der Bedarf da ist private Methoden zu testen, so deudet es auf ein Design Problem hin. In der XP Gemeinde ist dies aber allgemein umstritten, und auch meine ersten Erfahrungen widersprechen dieser These. (Ich habe bei weitem nicht die Erfahrung von Frank Westphal oder Kent Beck).

Es gibt ein Tool JUnitX mit der auch private Methoden getestet werden können.

2.3.5 Test First Ansatz

Es haben schon einige Firmen Erfahrung mit dem Test First Ansatz gemacht. Als Beispiel möchte ich aus dem Buch Test-Driven Development (TDD) von Kent Beck zitieren.

Can you test drive enormous systems?

Does TDD scale to extremely large systems? What new tests would you have to write? What you kinds of refactorings would you need?

The largest, totally test-driven system I've been involved with is at LifeWare (www.lifeware.ch). After 4 years and 40 person/years, the system contains approximately 250.000 lines of functional code and 250.000 lines of test code in Smalltalk. There are 4.000 tests, executing in under 20 minutes. The full suite is run several times each day. The amount of functionality in the system seems to have no bearing on the effectiveness of TDD. By eliminating duplication, you tend to create more smaller objects, and those objects can be tested in isolation independent of the size of the application.

Firmen in Deutschland, von denen ich weiss daß sie den Test-First Ansatz in ihren Projekten einsetzen (Diese Liste ist sicher sehr unvollständig):

- Andrena Objects Karlsruhe
- OIO Orientation in Objects Mannheim
- Daedalos

2.3.6 Test First Ansatz und einfacher Design

- Da wir zuerst die Tests schreiben, zwingen wir uns zunächst über die Schnittstellen des zu erstellenden Codes und erst später über die Implementation nachzudenken. Bzgl. Wartbarkeit ist die Qualität der Schnittstellen entscheidenter, als die eigentliche Implementation. Eine Änderung an einer Schnittstelle bedeudet, dass alle Objekte die diese Schnittstelle benutzen geändert werden muessen. Sind Schnittstellen stabil, bleiben Änderungen in der Implementation lokal.
- Unittests fördern das objektorientierte Denken. Eine Klasse wird als eigenständiges Programmteil gesehen.
- Da der Schwerpunkt auf Testbarkeit, und nicht alleinig auf Korrektheit steht, entstehen viele kleine Klassen, die weitgehend unabhängig voneinander sind, als grosse komplexe Klassen. Für komplexe Klassen ist es nämlich sehr viel schwieriger Unittests zu schreiben.
- Durch andauerndes Refactoring wird fachlich motivierter Code und rein technischer Code getrennt. Der rein technische Code (z.B. Mappen von Objekten in Datenbanken) kann auch in anderen Projekten wiederverwendet werden.

2.3.7 Literatur

- Kent Beck: Test Driven Development: ByExample
- Johannes Link: Unit Tests mit Java. Der Test First Ansatz
- Demnächst erscheint: Frank Westphal: Testgetriebene Entwicklung mit JUnit

2.4 Refacoring

2.4.1 Was ist Refactoring ?

Was ist guter Design

Hier meine Lieblingsdefinition aus einem Tutoriel über Extreme Programming

The right design for the software is one that

1. Runs all the tests
2. Has no duplicated logic.
3. States every intention important to the programmers.
4. Has the fewest possible classes and methods

In der obigen Definition ist die Reihenfolge der Anforderungen entscheident. Ein Wesen von Extreme Programming ist es, daß es keine Softwareanforderung gibt, die nicht durch Tests abgedeckt wird.

Not

- Most hooks
- Most abstract
- Designed for the ages

Was ist Refactoring

Durch Refactoring wird die interne Struktur von Software, aber nicht das externe Verhalten geändert. Das Ziel von Refactoring ist es den Design von Software zu verbessern. Dadurch verspricht man sich leichtere Wartung und Weiterentwicklung, sowie ein hoeherer Grad an Wiederverwendbarkeit

2.4.2 Warum führt redundanten Code zu höherer Wartung

Der beste Code ist nicht vorhandener Code

- Viele Entwickler ändern vorhandenen Code nicht, weil sie (zurecht) Angst haben existierende Funktionalitäten zu zerstören.
- Anstatt exisistierende Frameworks an neuen Anforderungen anzupassen, werden diese Frameworks weitgehend kopiert, und die Kopien den neuen Anforderungen angepasst. (Solange dem Entwickler die alten Anforderungen nicht bewusst sind, und er nicht weiss wie er die alten Anforderungen testen soll, ist dies der einzig machbare Weg sicher neue Anforderungen einzubauen).
- Der Entwickler benutzt nun die alten Frameworks für die alten Anforderungen, sowie etwas abgeänderten Kopien dieser Frameworks für neue Anforderungen, d.h. der Entwickler hat verschiedene Versionen desselben Codes

- Die Codemenge vergrössert sich, der Entwickler verliert den Überblick.
- Oft erkennt man dann nicht welche Teile des Frameworks in welchem Kontext gebraucht werden. Es existieren aehnliche Klassen, die sich im Klassennamen kaum unterscheiden (Beispiel: Teil1, Teil2, TeilDummy ; Nuplnutzer, Nupl_nutzer, Nutzplatinenutzer)
- Ändern sich alte Anforderungen, oder werden Fehler entdeckt, so müssen diese Änderungen sowohl an den kopierten als auch an den alten Frameworks vorgenommen werden.
- Je älter das Projekt wird, desto teurer wird die Wartung (auch fuer nur kleine Änderungen), desto frustrierter werden Entwickler und Kunde. Fluktuation der Mitarbeiter verstärkt diese Problematik.
- Zunächst verdienen wir an den hohen Wartungskosten sogar Geld. Allerdings bekommen wir erhebliche Probleme wenn der Entwickler wegläuft; bzw. der Kunde das Projekt stoppt bzw. an einer anderen Firma ueberträgt.

2.4.3 Warum entsteht redundanter Code

Redundanten Code kann viele Ursachen haben:

- fehlende Sensibilät der Programmierer in die Problematik
- Zeitdruck
- keine Qualitätssicherung vorhanden
- Programmierer kennen nicht die existierenden Frameworks
- kein Vertrauen in existierenden Code
- keine Möglichkeit die vorhandene Funktionalität zu testen

2.4.4 Wie entsteht durch Refactoring firmensspezifische Frameworks

- Es wird nur das codiert, was für die aktuelle Anforderung notwendig ist.
- Durch Refactoring wird redundanter Code gelöscht.
- Es wird automatisch fachlich und technischer Code getrennt, sobald es 2 verschiedene Anwendungsfälle gibt, die denselben technischen Code benötigen.
- Synergieeffekte zwischen verschiedenen Projekten, ist nur dann möglich wenn Wissen verteilt ist. Dies geschieht durch Pair Programming mit wechselnden Partnern.
- Verschiedene Projekte benötigen gemeinsames Repository

2.4.5 Ressourcen Refactoring

Refactoring Kenntnisse sollten zum Grundwissen jedes Programmierers von objekt orientierten Sprachen gehören. Hier einige Links um sich weiterzubilden.

- OO-Design Kurs von Klaus Meucht
- Die Refactoring Homepage

2.5 Programmierstandards

- Ein einheitlicher Programmierstil erleichtert die Wartung
- Programmierstandards sollten schriftlich festgehalten, und über Intranet verfügbar sein.
- Einhaltung von Programmierstandards nur durch gegenseitige Kontrolle möglich.
- Welcher Standard man wählt ist nicht so wichtig, wie daß der gewählte Standard einheitlich eingehalten wird.
- Es ist ausdrücklich erlaubt gegen Standards zu verstossen. Aber dann muss es im Code durch einen Kommentar begründet sein. Ggf. lohnt es sich sich mit anderen

Programmierern zu beraten.
- Im Laufe des Projekts koennen Standards geandert, geloescht und hinzugefuegt werden.
- Je nach Projekt können Standards unterschiedlich sein.

Typische Standards und Beispiele

- Formatierungskonventionen
  - Schliessende Klammer eines Blocks muss immer in derselben Hoehe der oeffnenden Klammer sein.
- Namenskonventionen
  - Alle Oberflächenklassen sollen mit dem Prefix UI anfangen.
  - Alle Methoden die einen booleschen Wert zurückgeben, und das Objekt selbst nicht verändern sollen mit einem Fragewort beginnen. (z.B istStatusBeendet)
- Dokumentationskonventionen
  - Methoden sollten i.a. nicht dokumentiert werden.
  - Eine Klasse soll mit max. 3 Saetzen beschrieben werden. Jede Instanzvariable sollte mit einem kurzen Satz beschrieben werden.
- Programmierkonventionen
  - Fange Exceptions nie mit leeren catch()Blöcken auf
  - Benutze Exceptions nur für unerwartete Fehler, die selten auftreten.
  - Benutze equals(), und nicht == um auf Gleichheit zu testen.
  - Nachdem Erzeugen eines Objekts mit Constructor, muss das Objekt immer in einem gültigen Zustand sein.
- Packagingkonventionen
  - Klassen die gegenseitig voneinander abhängig sind, sollten in einem Package sein.
  - Ein Package das sehr schwer veränderbar ist, sollte nicht von einem Package abhängig sein, das sehr leicht veränderbar ist.

Für die meisten Sprachen gibt es StyleGuides, die sich als sinnvoll erwiesen haben.

Beispiel Java:

- The Elements of Java Style
- Die Firma QA Systems hat Regeln und Patterns , die von Ihrem Produkt QStudio for Java automatisch ueberprueft werden, veröffentlicht.

Beispiel PLSQL: Oracle PL-SQL Best Practices

2.6 Architektur

2.6.1 Unterschiede einfacher Programmierer und Architekt

Programmierer

- Einziges entscheidentes Kriterium: Es funktioniert
- sucht die schnellste Lösung
- fügt i.a. immer neuen Code hinzu, ändert oder löscht i.a. nicht bestehenden Code

Architekt

- Wichtigstes Kriterium: Es funktioniert
- Vermeidet Redundanzen ==> Refactoring
- sucht die einfachste Lösung (dies ist i.a. nicht die schnellste Lösung)

- kennt die typischen Programmierstandards und Patterns und achtet das diese eingehalten werden.
- Achtet auf [qualitativ hochwertige Schnittstellen > http://urmel.ivm-solve-it.com/docs/ooDesignUpd/schnittstellen.htm#schnittstellen]
- Versucht Abhängigkeiten zu vermeiden, besonders gegenseitige Abhaengigkeiten.

Meine Forderung:

Wir sollten alle Programmierer zu Architekten schulen

2.6.2 Vergleich Software Architektur mit einem Unternehmen

Mit dem Wachstum der Mitarbeiteranzahl (und aus anderen Gruenden), oder neuen Projekten organisert ein Unternehmen ihre Abteilungen neu. Man kann diese Umorganisation auch als Refactoring bezeichnen.

Das Ziel der Umorganisation:

- Kleine uebersichtliche Einheiten, um Probleme schneller zu erkennen.
- Kommunikation vereinfachen da
  - Mitarbeiter die eng miteinander zusammenarbeiten, in einer Abteilung sind.
  - Mitarbeiter die eng zusammenarbeiten, in der Nähe sitzen
  - Besprechungen sind im kleineren Kreis, damit kürzer und zielgerichteter.
- Redundante Dienste auslagern
  - Es gibt Dienste, die alle (produktiven)Abteilungen brauchen, diese sind in einer eigenen Abteilung (Verwaltung) zusammengefasst. Die Verwaltung ist ein Dienstleiter fuer alle anderen Abteilungen. Jede Abteilung seine eigenen Systemadministratoren bzw. jede Abteilung sein eigenes Personalbüro wäre redundant
- Die Verwaltung sollte sowenig möglich Wissen der Abteilungen benötigen, damit sie sich auf Ihre eigentlichen Aufgaben konzentrieren können. Die Mitarbeiter der einzelnen Abteilungen sollten sowenig Verwaltungsaufwand wie moeglich haben.(z.B nicht gleichzeitig Stunden in Excel und SAP bzw. Navision eintragen).
- Die einzelnen (produktiven) Abteilungen, sind i.a. nicht voneinander abhaengig, und können eigenständig arbeiten. (Dies stimmt nicht ganz, da Mitarbeiter an andere Abteilungen ausgeliehen werden. (Kappa-Show). Objekte und Mitarbeiter lassen sich zum Glück nicht ganz vergleichen).
- Sind die Abteilungen voneinander unabhaengig, so kann der Unternehmer einzelne Abteilungen auslagern. (Ich hoffe dem Unternehmer ist dabei bewusst, dass er es mit Menschen und nicht mit Objekten zu tun hat)
- Wichtig: Da sich der Markt dauernd ändert, sind immer wieder Umorganisationen notwendig.

Software Architekten machen i.a. genau dasselbe:

- Objekte die eng miteinander zusammenarbeiten werden in gemeinsame Abteilungen (Paketen) gehalten.
- Redundante Aufgaben von mehreren Abteilungen werden herausgenommen, und von allen Abteilungen genutzt.
- Wird eine einzelne Abteilung (Paket) zu gross, so dass sie nicht mehr von einer Person überschaut werden kann, versucht man das Paket in kleinere Paketen aufzuteilen.
- Abhaengigkeiten zwischen den Paketen werden minimiert. Besonders gegenseitige Abhaengigkeiten werden vermieden. (Verwaltung ist Dienstleister fuer die produktiven Abteilungen und nicht umgekehrt).

- Da die Pakete weitgehend voneinander unabhängig sind, sind bei neuen A nforderungen nur wenige Pakete davon betroffen. (Wenn ein Unternehmen eine neue Abteilung aufmacht; sollten nur die Mitarbeiter der neuen Abteilung sowie vielleicht noch die Verwaltung davon betroffen sein).
- Da die Pakete voneinander weitgehend unabhaengig sind, ist es einfach diese z.B vom Clienten zum Server zu verschieben (oder umgekehrt).

Bermerkungen dazu:

- So wie ein Unternehmer auf die Kommunikationsfluesse in seinem Unternehmen achtet, achtet der Software-Architekt auf die Kommunikationsfluesse zwischen den Objekten. Hohe Kommunikation ist wünschenswert, aber sie muss strukturiert und zielgerichtet sein.
- Ein Unternehmer versucht Redundanzen zu vermeiden, da jeder Mitarbeiter Geld kostet. Dem Programmierer ist leider oft nicht bewusst, dass redundanter Code Geld kostet. Der Programmierer fügt bei Bedarf eben einfach neue Objekte hinzu, und schaut nicht ob schon an anderere Stelle eine Loesung existiert. Würde ein Unternehmer so handeln, dann hätte z.B. jede Abteilung sein eigenes Personalbüro. Eine Abteilung würde seine Stunden in SAP, die andere in Navision, und wieder eine andere mit beiden Systemen ihre Stunden verwalten. (Und irgendein armes Schwein muss die Ergebnisse aus allen Systemen zusammenfassen).

2.6.3 Zusammenhang Architektur Patterns und Refactoring

Ziel einer guten Architektur ist es schnell auf Änderungen reagieren zu können.

Als Mass fuer die Produktivitaet eines Softwareunternehmens, kann die Geschwindigkeit angesehen werden; in der neue Features oder Änderungen eingebaut werden.

Bei einer schlechten Architektur sind viele Module (Pakete) voneinander abhaengig. Eine Aenderung erzwingt oft, in vielen Modulen eingreifen zu muessen.

Bei einer guten Architektur ist die Wahrscheinlichkeit sehr hoch, die Änderung nur in einem Modul durchfuehren zu muessen. D.h die Architektur eines Systems beeinflusst direkt die Produktivitaet.

Refactoring ist nichts anderes als die Verbesserung der Architektur. Ein Softwareunternehmen das auf Refactoring verzichtet, ist wie ein Holzfaeller der auf das Schaerfen der Axt verzichtet, da man dafür kein Geld bekommt.

Kenntnisse in (Design) Patterns erleichtern es schnell eine geeignete Architektur zu finden. Es ist aber auffällig dass man durch Refactoring automatisch die Patterns enthält.

Es gibt einen sehr interessanten Artikel Was Software Entwicklung mit der Thermodynamik verbindet von Jens Coldewey aus der Zeitschrift Objekt Spektrum der aufzeigt, wie durch konsequentes Refactoring die Komplexitaet von Software verringert und die Produktivität erhöht wird.

2.7 Versions- und Konfigurationsmanagement

Anforderung an ein Versions und Konfigurationsmanagement Warum so oft wie moeglich integrieren.

Versionierung:

- Entwickler haben i.a. neuere Versionen als die Anwender
- Wenn Anwender Fehler melden, sollte man den Fehler in der Version die der Anwender hat nachvollziehen.
- Jeder macht Fehler oder anderst ausgedrückt wer keine Fehler macht arbeitet nicht. Um erfolgreich zu sein braucht man den Mut Fehler zu machen. Wenn man versioniert hat, also leicht auf einen korrekten Stand zurückgehen kann; hat man auch den Mut ungewöhnliche Lösungen auszuprobieren.
- Es gibt i.a. nichts was sich nicht lohnt zu versionieren. Anforderungen, technische Konzepte, Testskripts, Besprechungsprotokolle und natürlich Source Code.
- Allgemeines Versionierungstool: **CVS**
- Versionierung von Datenbanken hat sich als sehr schwer erwiesen
  - Hat man verschiedene Datenbanken (Test, Entwicklung, Produktion) so besteht die Gefahr, daß die bestehenden Datenbanken auseinanderlaufen.
  - Erlaubt man Änderungen an den Datenbanken nur durch eine Person, so entsteht die Gefahr eines Flaschenhalses. Änderungen an Datenbank müssen beantragt werden.
  - Hat jeder Zugriff auf jede Datenbank, so entsteht sehr schnell Chaos. Versionierung und Integration nur sehr schwer möglich.
- Mögliche Lösung zur Versionierung von Datenbanken:
  - Änderungen an Datenbanken sollten nur durch Skripte erfolgen.
  - Es gibt 3 Arten von Datenbanken: Gold, Silber und Bronze.
    - Die goldene Datenbank ist die Produktionsdatenbank. Die goldene Datenbank wird i.a. nur von einer Person gewartet.
    - Die silberne Datenbank entspricht einem einheitlichen Stand auf dem Weg zur goldenen Datenbank. Sie entspricht immer der letzten Integration der bronzenen Datenbanken. Die silberne Datenbank wird i.a. von nur einer Person gewartet.
    - Jeder Entwickler hat seine eigene Bronze Datenbank. Dort kann er darin beliebig experimentieren. Einmal in der Woche werden die Bronze Datenbanken der Entwickler zu einer silbern Datenbank zusammengeführt. Danach bekommen die Entwickler eine Kopie der silbernen Datenbank als neue bronzene Datenbank.

Integration

- Wenn mehrere Personen an einem Projekt arbeiten, erhöht sich mit jeder Änderung die Gefahr; daß man auf alten Projektstand aufbaut.
- Ziel: So oft wie möglich die Arbeiten der Mitarbeiter integrieren; am besten täglich.
- Nach jeder Integration müssen alle Unittests erfolgreich laufen. Ggf. werden nach jeder Integration automisierte Qualitätskontrollen durchgeführt (Code Metriken). Diese Qualitätskontrollen sollten die Entwickler aber schon selber vor der Integration machen.
- Sind ein oder mehrere Unittests fehlerhaft, so muss zuerst der Fehler korrigiert werden, bevor jemand weiterentwickelt. Im Notfall muss man Code wegschmeissen; um wieder einen stabilen Stand zu bekommen.
- Nach jeder erfolgreichen Integration arbeiten die einzelnen Entwickler an einer Kopie der integrierten Version. Damit wird garantiert dass alle Entwickler am Anfang des Tages auf dem aktuellen Stand aufbauen.

Konfiguration

- Nicht jeder Anwender braucht alle Funktionalitäten des aktuellen Entwicklungsstand. Konfigurationsmanagement ermöglicht es aus dem Entwicklungsimage verschiedene RuntimeImage zu bauen. Es können gewisse Funktionalitäten eingebaut bzw. ausgebaut oder ausgetauscht werden
- Beispiele:
  - Aus einem Entwicklungsimage werden für jedes Betriebssystem ein eigenes Runtime

Image gebaut.
    - Aus einem Entwicklunsgimage wird ein eigenes Runtime Image für Client und Server gebaut.
    - Es werden je nach verwendeter Datenbank verschiedene Runtime Image benötigt (Ein Kunde hat Oracle der andere MySQL).
- Ohne durchdachte Softwarearchitektur ist i.a. keine Konfiguration möglich. Man muss wissen welche Softwarepakete voneinander abhängig sind; und es sollten keine gegenseitigen Abhängigkeiten zwischen Softwarepaketen sein.
- Konfigurations Tool fuer Java: ANT

2.8 Projektverfolgung

Der Projektzustand muss messbar sein

Was will ich verfolgen?

- Aktueller Stand des Projekts
    - Welche Aufgaben sind offen, bearbeitet und getestet.
    - Verhindern von vielen offenen Punkten (80 zu 20% Faktor).
    - Vergleich mit den Anforderungen muss möglich sein.
- Wer arbeitet gerade in welchem Bereich
    - Aufgabe schriftlich formulieren, bevor sie gelöst werden.
    - Release Notes nach der Arbeit
- Welcher Bereich verursacht welche Kosten ?
    - Stundenzettel ehrlich ausfüllen
    - Geschätzte und wirkliche Kosten eintragen.

Ein einfaches, preiswertes aber geniales Projektverfolgungstool hatte ich in einem auswärtigen Projekt kennengelernt.

Es bestand lediglich aus einem FlipChart:

Auf diesem FlipChart standen alle offenen und zuletzt abgeschlossene Punkte. Dieses FlipChart stand zentral in einem Raum, für jeden einsichtbar. Die Punkte waren nur stichwortartig. Darunter stand welche beiden Programmierer diesen Punkt bearbeiten, bzw. bearbeiteten und wie gross der geschätzte Aufwand ist, bzw. wie lange daran schon gearbeitet worden ist.

Einmal in der Woche habe sich alle getroffen, um das Flipchart zu aktualisieren. Die Schätzungen sind aktualisiert worden. Neue offene Punkte sind eingetragen, bzw einige Punkte die nicht mehr aktuell waren entfernt worden.Die Entwickler haben dann selbstständig sich geeinigt wer welche Punkte mit wem in der nächsten Woche bearbeiten soll.

Diese Vorgehensweise hatte einige Vorteile. Die Entwickler sahen nicht nur ihr Spezialgebiet sondern den Gesamtzusammenhang in einem Projekt. Jeder hatte auch die Möglichkeit sich in Bereiche einzuarbeiten, die für ihn neu waren. Meistens suchte er sich dann einen Partner aus, der sich in diesem Gebiet sehr gut auskannte. So war der Lernfaktor in diesem Projekt sehr gross.

2.9 Pair Programming und Collective Code Ownership

2.9.1 Was ist Pair Programming?

Aus der Homepage von Frank Westphal :

Die Regel lautet: Wer um Hilfe bittet, dem wird Hilfe geboten. Tatsächlich wird keine Zeile Produktionscode geschrieben, ohne daß zwei Paar Augen auf den Bildschirm gerichtet sind. Das bedeutet, Sie programmieren zu zweit an einem Rechnern mit einer Tastatur und einer Maus. Sie sitzen nebeneinander, führen ein intensives Gespräch über den entstehenden Code und wechseln sich regelmässig an der Tastatur ab. Sie dürfen dabei sogar Spaß haben, denn Programmieren soll Spaß bringen. Sie wechseln häufiger Ihre Programmierpartner und am Ende der Woche haben Sie idealerweise mal mit jedem Ihrer Kollegen zusammengearbeitet, damit sich das aufgebaute Wissen über das gesamte Team verbreitet.

Bemerkungen:

- Ich habe Erfahrung mit wöchentlichen Tausch des Partners gemacht. Die Dauer der sinnvollen Zusammenarbeit hängt davon ab, wie stark die Aufgaben in Teilaufgaben gebrochen werden können.
- Die Programmierer einigen sich selber darauf wer mit wem zusammenarbeitet.
- Pair-Programming ist i.a. anstrengender als alleine zu programmieren, macht aber mehr Spass. Die meisten die es ausprobiert haben, wollen es nicht mehr missen.
- Durch Pair Programming eignet sich der einzelne Programmierer sehr schnell Wissen an.

2.9.2 Was ist Collective Code Ownership

Der gesamte Code gehört dem Team. Jedes Paar soll jede Möglichkeit zur Codeverbesserung jederzeit wahrnehmen. Das ist kein Recht sondern eine Pflicht. Da die Funktionalitaet mit automatisierten Tests abgedeckt ist, ist die Gefahr dass das Programm bei der Codeverbesserung zerstört wird, gering. Eine weitere Sicherheit ergibt sich durch die Versionierung von Code.

Collective Code Ownership hilft den Code so zu schreiben, dass er selbst kommentierend ist.

2.9.3 Gefährlichkeit von genialen Lösungen

Ich habe in einige Projekte Entwickler getroffen, die sehr schnell und auch fehlerfrei entwickelten. Oft haben gerade diese Leute dem Projekt mehr geschadet als genutzt.

- Diese Entwickler haben oft die existierenden Frameworks nicht benutzt, und "by the fly" eigene entwickelt.
- Management Fehler sind oft überdeckt worden. Da die Anforderung unklar sind, verwenden diese Entwickler oft "Goldrandloesungen"
- Der Erfolg von einem Projekt hängt von wenigen Mitarbeitern ab. Ein Projekt sollte immer in einem Zustand sein, dass jeder Mitarbeiter austauschbar ist.
- Der Code ist von weniger genialen Mitarbeiter schwer zu warten

Die Loesungen von heute, sind oft die Probleme von morgen

2.9.4 Vorteile Pair Programming und Collective Code Ownership

- schnelle Wissensverbreitung
- hoher Lernfaktor
- geringer Frustrationsfaktor (Never walk alone gegen Jeder wurschtelt alleine fuer sich rum).

- gleichmaessigere Auslastung

2.9.5 Wissenschaftliche Studien

Pair Programming kann auch ausserhalb von Extreme Programming angewendet werden. Es gibt einige Forschungsarbeiten, die den Erfolg von Pair Programming beweisen.

Diese Forschungsarbeiten sind hier veroeffentlicht.

Siehe speziell auch den Artikel von Laurie Williams und Alistair Cockburn The Costs and Benefits of Pair Programming

- Geringere Abhängigkeiten von einzelnen Mitarbeitern.
- Möglichkeiten in Urlaub zu gehen.
- Synergieeffekte nutzen (Ein ähnliches Problem hatten wir ja schon gelöst)

3 Die Werkzeuge klein aber mächtig

3.1 Junit

JUnit ist das zentrale Tools für Unittests. Auf der Homepage von JUnit sind eine Vielzahl von Artikel über TestFirst Programming und viele zusätzliche Tools, die Junit ergänzen.

3.2 JDepend

JDepend ist ein kostenlosses Tool das Java Source Code auf PaketEbene analysiert. Es misst die Abhängigkeiten zwischen Paketen und erkennt auch zyklische Abhängigkeiten.

3.3 CodeCoverage

Tools die den Source Code analysieren, sind eine gute Ergänzung zu manueller Codeinspektion. Beide Arten von Codeinspektionen können sich aber gegenseitig nicht ersetzen.

Das Ergebnis von automatisierten Codekritiken können Akzeptanzkriterien sein, und dies kann direkt in einem Vertrag hineingeschrieben werden. Allerding muss man einige Toleranzen offen halten. Nicht jede Kritik die ein Code Analyse Tool findet ist berechtigt, in einigen seltenen Fällen ist es auch sinnvoll bewusst gegen typische Regeln zu vertossen.

Wir können Code-Qualität garantieren und mit diesen Tools messbar nachweisen

In Java gibt es viele kommerzielle (und teure) Tools die den Source Code analysieren. Falls der Kunde diese besitzt und zur Verfügung stellt, sollten wir sie nutzen.

Inzwischen gibt es auch erste OpenSource Tools:

- PMD

Auszug aus der Beschreibung:

PMD scans Java source code and looks for potential problems like:

- Unused local variables
- Empty catch blocks
- Unused parameters
- Empty 'if' statements
- Duplicate import statements
- Unused private methods
- Classes which could be Singletons
- Short/long variable and method names

PMD gibt es unter anderem fuer Eclipse, Forte und JBuilder

- CPD

CPD ist ein Copy Paste Detector und baut auf PMD auf. Er sucht im Java Code nach identischen Code (fragmenten).

- Checktyle (Link muss noch eingetragen werden).

3.4 CoverageAnalyse

Eine Coverage Analyse gibt Hinweise auf Vollständigkeit der Tests. Eine Coverage Analyse protokolliert welcher Code bei einem Test mindestens einmal bzw. nie durchlaufen wurde.

Eine Coverage Analyse hilft toten Code und Lücken in den Tests zu finden. Man beachte daß 100% Testabdeckung noch keine Garantie für die Korrektheit des Programms gibt:

Test State Coverage, Not Code Coverage

<pre> int test(int a, int b){ return a / (a + b); } </pre>

In der XP-Community wird häufig Clover empfohlen. Es ist leider nicht mehr Open Source. Das Preis, Leistungsverhältnis soll aber sehr gut sein.

Die Einsatzmöglichkeiten von Clover beschreibt auch ein deutschsprachiger Artikel der Firma OIO

Ein anderer Weg geht Jester . Jester ändert einfach den Code und lässt alle Tests durchlaufen. Wenn immer noch alle Tests korrekt durchlaufen, wird der Teil des Codes den Jester geändert hat nicht abgedeckt. Jester ist Open Source

3.5 Versionsverwaltung

Das bekannteste ist sicherlich CVS . Es lässt sich auch direkt in Eclipse einbinden.

3.6 Fehlerdatenbank

Ziel einer Fehlerdatenbank:

- Nachverfolgen welche Fehler und Features implementiert sind
- Ermitteln offener Punkte mit Priorisierung
- Zuweisung von Aufgaben an Verantwortliche
- ToDo Listen fuer die Mitarbeiter.

- Statistische Analyse welche Art von Fehler wie häufig vorkommt.
- Nachvollziehen, wer welche Lösung aufgrund welcher Anforderungen bzw. Problem implementiert hat.
- Dijenigen die ein Fehler melden bzw. Feature beantragen sollen garantiert auch benachrichtigt werden.
- Überblick bewahren, wer an welchem Problem arbeitet.
- automatische E-Mail Benachrichtigung, wenn sich in dieser Liste etwas aendert
- automatische Erinnerung wenn Punkte offen bleiben.
- Verschiedene Sichten für Kunde und Softwarehaus. Es muss möglich sein Bemerkungen hineinzufügen, die der Kunde nicht lesen kann.
- Volltextsuche über Stichworte

Tool - Vorschlag:

- Ein Tool das von der Beschreibung aus sehr brauchbar zu sein scheint ist FogBUGZ . Es ist leider nicht kostenlos, aber mit unter 1000 Dollar fuer 10 Personen, kostet es nicht die Welt. Vom Autor dieser Software ein netter Artikel über Fehlerverwaltung.
- Ein bekanntes Open Source Tool ist Bugzilla . Hier ist eine deutschsprachige Beschreibung und Installationshilfen.