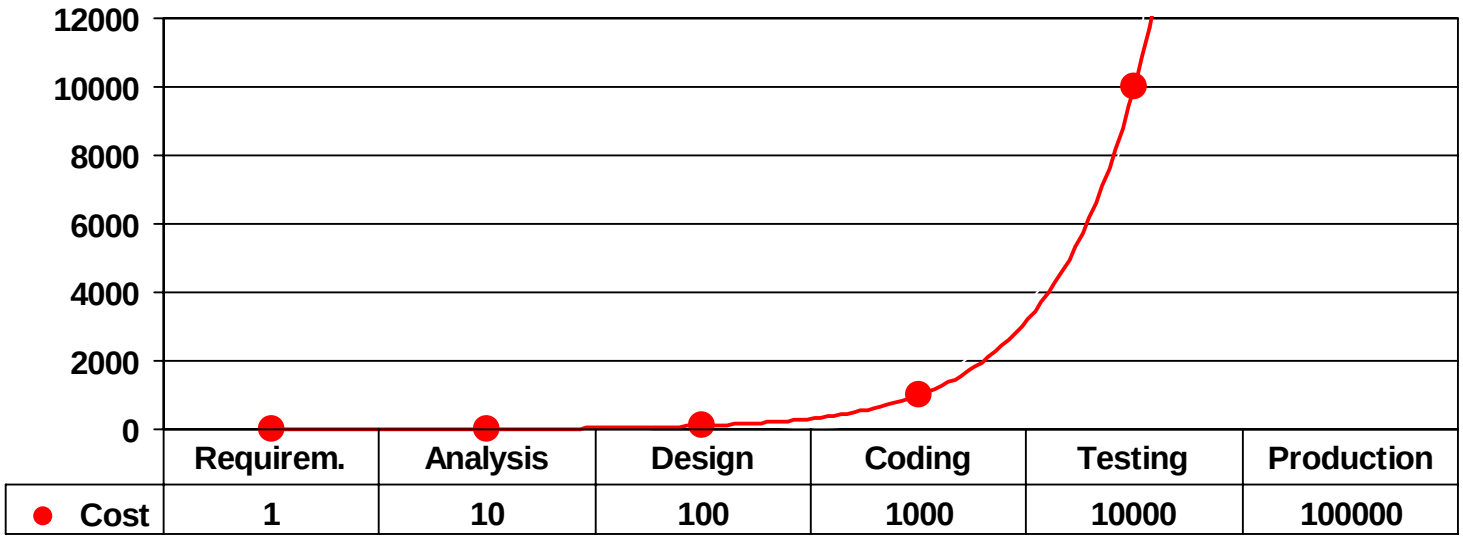
Iterative Softwareentwicklung

Ullrich Köthe

19. April 2001

Traditionelles Entwicklungsmodell

- **von Geräteentwicklung inspiriert: je später eine Änderung notwendig wird, desto teurer**



	Requirem.	Analysis	Design	Coding	Testing	Production
Cost	1	10	100	1000	10000	100000

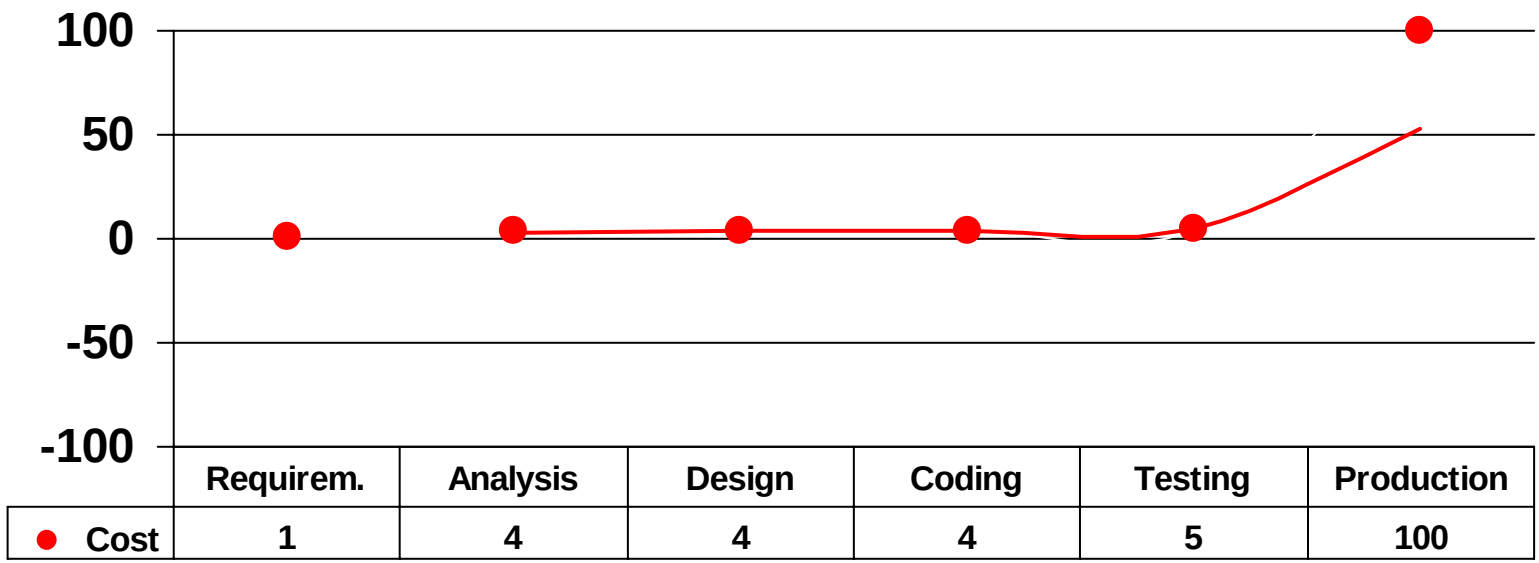
⇒ Änderungen möglichst früh durchführen
Fehler möglichst früh finden

Konsequenzen

- **Treffen aller wichtigen Entscheidungen vor Beginn der Implementation**
  - formale Analyse und Design (CASE-Tools)
- **aber:**
  - hohe Komplexität des Entwurfs
  - viel Papier statt Software
  - große Bürokratie $\Rightarrow$ große Teams, geringe Flexibilität
  - unnatürliche Arbeitsweise, geringe Motivation
- **und: späte Änderungen trotzdem nicht zu verhindern**
  - neue Anforderungen, neue Umgebung !
  - Fehler bei Analyse/Design unvermeidlich

Erstrebenswertes Kostenmodell

- **Kosten pro Änderung steigen nur langsam:**



	Requirem.	Analysis	Design	Coding	Testing	Production
Cost	1	4	4	4	5	100

⇨ **Iterative Softwareentwicklung möglich**

Iterative Softwareentwicklung

- **allmählicher Aufbau eines Systems**
  - Zerlegung der Funktionalität in Häppchen, die nacheinander hinzugefügt werden
  - System ist jederzeit lauffähig, wenn auch mit zunächst geringer Funktionalität
- **Änderungen des Vorhandenen ist alltäglich**

⇨ **Nutzung des „weichen“ Charakters von SW**

  - flexible Planung, Anpassung an Kundenwünsche und veränderliches Umfeld
  - ständiges Dazulernen, keine voreiligen Entscheidungen

Warum sind Änderungen teuer?

- **unverständlicher Code**
- **veraltete Dokumentation**
- **nicht-lokale Änderungen, Folgeänderungen**
- **Änderungen sind Fehlerquelle (“Never change a running program!”)**
- **Bürokratie (Code-Ownership, Aktualisierung der internen Dokumentation etc.)**

⇨ **nicht nur Funktionalität, sondern auch Codestruktur muß laufend angepaßt werden -**

„Refactoring“

Konstante Kosten für Änderungen: Hauptsätze der Iterativen Entwicklung

1. HS: Verbesserung der Codestruktur (Refactoring) und Erweiterung der Funktionalität sind gleich wichtig

2. HS: Refactoring und Erweiterung müssen immer abwechselnd, *niemals* gleichzeitig, ausgeführt werden

- ⇨ **Refactoring: Änderung der Codestruktur *ohne* Änderung der Funktionalität**
- ⇨ **Erweiterung: Hinzufügen neuer Funktionalität *ohne* Änderung der Codestruktur**

Erweiterung ohne Strukturänderung

- **Wie soll das gehen?**
- **Ändern der Funktionalität immer als Erweiterung: Hinzufügen einer neuen Alternative an einem vorhandenen Variationspunkt, z.B:**
  - neue Implementation eines vorhandenen Interface
- **Wenn kein geeigneter Variationspunkt vorhanden**
  - ⇨ **zuerst Refactoring, um einen Variationspunkt zu schaffen, z.B:**
  - Einführen eines neuen Interface mit zunächst nur einer Implementation (vorhandene Funktionalität), erst danach Hinzufügen der neuen Funktion wie oben

Ziele von Refactoring

- **Vorbereiten des Codes für nachfolgende Erweiterungen**
  - **Einfügen von Variationspunkten**
- **Aufräumen, wenn Code schlecht strukturiert**
  - **Kapselung**
  - **Beseitigung von Redundanz**
  - **Beseitigung unnötiger (z.B. zyklischer) Abhängigkeiten**
- **Selbstdokumentierender Code**
  - **aussagekräftige Namen**
  - **verständliche Algorithmen**
- **Verbesserung der Rechengeschwindigkeit**

Erhaltung der Funktionalität beim Refactoring

Erhaltung der Funktionalität (Semantik) ist entscheidend:

- **ermöglicht laufende Kontrolle des Refactoring mittels Testprogrammen**
- **sichert, daß vorhandene Funktionalität nicht zerstört wird („Never change...“ gilt nicht mehr)**
- **gleichzeitige Änderung der Funktionalität würde Tests aushebeln ⇨ Fehler unvermeidbar**
- **ermöglicht „Aggressivität“ ⇨ optimale Struktur kann auch in schwierigen Fällen erreicht werden**

Wie soll man refaktorisieren?

- **Katalog von Standard-Refactorings (Buch: Fowler: „Refactoring“)**
  - **Erzeuge Funktion für mehrfach benutzten Code**
  - **Überführen einer Methode in andere Klasse**
  - **Schaffen einer Klasse für Gruppe von Attributen**
  - **~70 weitere**
- **Design Patterns (bewährte Strukturen für typische Probleme, Buch Gamma et al.)**
  - **Composite: Variationspunkte für Knoten eines Baumes**
  - **Iterator: VP für Typ einer Datenstruktur**
  - **Strategy: VP für Algorithmus**
- **Redundanzbeseitigung im konkreten Kontext**

Beispiel Refactoring

```
void foo(double amount, int currency)
{
    ...
}
```

Schritt 1: Kapseln von Variablen in einer Klasse

```
void foo(Money money)
{
    ...

}
```

```
class Money
{
     static final int EUR = 1;
     static final int USD = 2;
     double amount;
     int currency;
};
```

Schritt 2: Erzeugen einer Basisklasse

```
abstract class Money {...};
class Euro extends Money
{...};
```

Was ist zu beachten ?

- **Refactoring sofort, wenn etwas nicht stimmt**
- **umfangreiches Refactoring schrittweise, über eventuell langen Zeitraum (im Wechsel mit Erweiterungen)**
- **Zwischenergebnisse werden oft im nächsten Schritt wieder geändert**
  aber: Zwischenschritte sind notwendig zur Erhaltung der Funktionalität
- **Unit Tests kontrollieren Refactoring**
- **Erfahrung hilft bei der Erkennung der Notwendigkeit von Refactoring und bei der Planung des Vorgehens**

Unit Tests

- **100% Regel: unit tests müssen 100% funktionieren**
- **wenn unit tests unter 100%: sofort reparieren (letztes Refactoring rückgängig machen)**
- **häufiges Ausführen der Testsuite (nach jeder Änderung)**
- **schreiben eines Tests *vor* der Implementation einer Funktionalität (oder gleichzeitig)**
- **bis zu 50% des Codes ist Testcode!**

Organisches Wachstum der Code-Struktur

- weniger „Big Design Upfront“, dafür Design-Entscheidungen wenn nötig („Just-in-Time“)
  - maximales Wissen über Problem, Lösung und Kontext
  - das einfachste mögliche Design für aktuellen Systemausbau
  - Design paßt sich im Laufe der Systementwicklung organisch an
  - ständiges Refactoring (nicht nur alle 6 Monate)
- System Metaphor: übergreifendes Architekturkonzept für Konsistenz des Systems (das große Ganze im Auge behalten)

Iterativer Entwicklungsprozeß

Formulieren der Funktionalität in User stories

- **maximal 3 Tage pro User Story**
- **Ordnen der Stories nach Priorität - grundlegende und kritische Stories zuerst**
- **nicht implementierte Stories jederzeit zu entfernen oder zu ändern, neue hinzuzufügen**
- **kurze Inkremente, stets funktionierendes System ⇨ “piecemeal growth”**
- **Fortschrittskontrolle anhand des Load factors**
- **kein allmähliches Verschlechtern der Systemstruktur durch Refactoring**