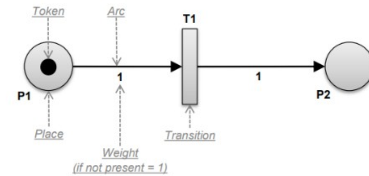
Softwaretechnologie 2

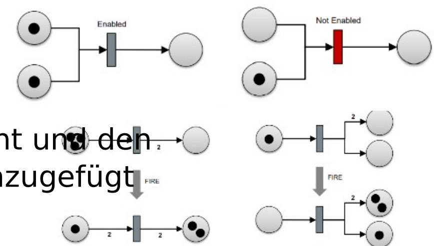


1. Petri-Netze (PN):

- Gerichteter, bidirektionaler Graph über 2 Arten von Knoten: Plätze, Transitionen

Elementar Nets *(Predicate/Transition Nets)*	➢ PN mit Bool'schen-Tokens *(max. 1 Token pro Platz)* - Plätze = Bedingungen / Zustände / Prädikate - Transitionen = Feuern von Ereignissen
Integer Nets *(Place/Transition Nets)*	➢ Gewichtetes PN mit beliebig vielen Integer-Tokens
High Level Nets	➢ Reduzibles, modulares PN - Plätze = Zustände; Transitionen = Aktionen

- **Transition**
  - **aktiv:** Anzahl eingehend. Tokens gleich der Gewichtung eingehender Arcs
  - **feuert:** Token wird entspr. Der Gewichtung entfernt und den ausgehenden Plätzen entspr. Deren Gewichtung hinzugefügt



- **Verfeinerung durch Modularisierung:** Unternetz wird Seite *(Modul)* genannt

Transition-Unternetz	Platz-Unternetz
Schnittstelle besteht aus Transitionen	Schnittstelle besteht aus Plätzen

- **Eigenschaften:**
  - **Erreichbarkeit:**
    - ➢ Markierung (Platz) M_n erreichbar von M_0, wenn es feuernde Sequenz $s = t_0 \ldots t_n$ gibt
    - $R(M_0)$: Menge aller erreichbaren Markierungen aus M_0
  - **Begrenztheit *(k-Boundedness)*:** Plätze haben maximal k – Tokens; PN sicher mit $k=1$
  - **Lebendigkeit:**
    - **PN:** lebendig $\rightarrow$ alle Transitionen L4
    - **Transition:** drückt aus, ob Transition aktiv bleibt oder nicht, ausgehend von M_0
      - L0: Transition nicht feuerbar
      - L1: Transition kann mind. 1 Mal gefeuert werden
      - L2: Transition kann mind. k Mal gefeuert werden
      - L3: Transition unendlich oft gefeuert in manch Sequenzen
      - L4: Transition von jeder Markierung/Platz aus mind. 1 Mal feuerbar
    - **Markierung:** Dead, wenn keiner seiner Transitionen aktiv sind
- **Incidence (Transition) Matrix A:**
  - ➢ Repräsentiert PN in Matrixform
  - ➢ Zum Testen von Lebendigkeit eines PN:

1. Minimale T-Invarianten *(x-Vektor aus $A^T x = 0$, sodass $M_i \rightarrow M_i$ abgebildet)* berechnen
2. Berechne Switch-Vektor *(Summe aller min. T-Invarianten)* & überprüfe auf Nichtvorhandensein von 0-Einträgen
3. Baue Erreichbarkeitsgraphen und überprüfe, ob M_0 von jeder erreichbaren Konfiguration erreichbar ist

2. Anforderungsanalyse:

- **Regel:** Arbeite problemorientiert - von Probleme zu Ziele & Anforderungen

- **Ablauf:**

1. Vorstudie	- **Lastenheft:** Projektziel, Zielgruppen, grobe Anforderungen, Funktionen, Fristen - **Kostenanalyse:** Thematisiert fixe & variable Kosten, Nebenkosten *(Equipment, Reise)*, Zeit - **Risikoanalyse:** Verhindert Fehleinschätzung der Kosten & Fristen ➢ **Resultat:** Risikomanagementplan & Risikoliste - **Projektplan:** Gesamtheit aller im Projekt vorhandenen Pläne
2. Kontextanalyse	- Benutzeranalyse - Domainanalyse *(Begriffshierarchie mit Beziehungen & Einschränkungen)* - Problemanalyse - Zielanalyse
3. Anforderungs-spezifizierung	- Funktionale Anforderungen - Nichtfunktionale Anforderungen - GUI Prototype
4. Systemanalyse	- Kontextmodel *(Beschreibt Schnittstellen zur Außenwelt)* - Top-Level Architektur *(Zeigt Interaktion der Schnittstellen mit den Teilkomponente)*

- **ZOPP - Entwicklungsprozess:**
  - ➢ Zielorientierte Projektplanung mittels hierarchische Problem-Ziel-Funktion Analyse

1. **Benutzeranalyse**
2. **Problemanalyse (Ist-Analyse):** Ermittlung der Probleme & Nutzungsgründe
   - **Phase 1:** Problemsammlung (Brainstorming)
     Problembaum: Hauptproblem mit zugehörigen Subproblemen
   - **Phase 2:** Ursachenermittlung der Probleme mittels *Root-Cause-Graph*
     (Probleme als Fragen mit entspr. Ursache)
     Konsequenzermittlung mittels *Cause-Effect-Graph*
     (Problembaum mit zugehörigen Effekten der Probleme; getrennte Bäume)
   - **Phase 3:** Priorisierung der Subprobleme
3. **Zielanalyse (Soll-Analyse):** Ableitung der Ziele aus Ursachen & Folgen
   - ➢ Abhängigkeitsgraph der Ziele

4. **Funktionale Anforderungsanalyse:** Was soll es können; bspw. mit Funktionsbaum
5. **Nicht-funktionale Anforderungsanalyse:** Qualitätsanforderungen
6. **Akzeptanzkriterienanalyse:**
   - Beschreibung aller verpflichtenden, messbaren non-/semi-/funktionale Anforderungen zur Erfüllung des Projektes
7. **Akzeptanztests:** zeigt Erfüllung der Akzeptanzkriterien an

- **Anforderungs-Management:**
  - Weiterentwicklung der SRS *(Beschreibt gewünschte Merkmale des Systems)* mit Kunden
  - **Anforderungsmanagementsystem:** Nutzung einer DB, die Anforderungen enthält

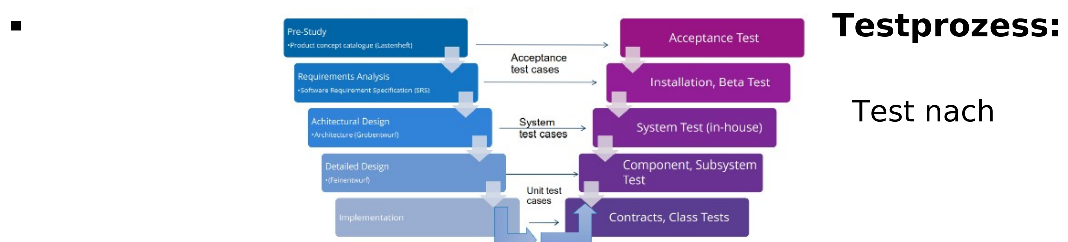
3.Validierung *(Testen auf korrektes Verhalten)*:

- **Durch Defensive Programmierung:**
  - Programmierung, sodass so wenig wie möglich Fehler entstehen
  - **Vertragsprüfung durch Schichten um Prozeduren:**
    - **Assertions:** Spezifizieren Tests in Prozeduren
    - **Pre-/Postcondition Checks**
    - **Invariant Checks:** Form der Datenstruktur
    - **Pattern:** Contract Wrapper Layer *(Pre-/Postcondition Checks um Prozedure)*

- **Mit Inspektion and Bewertungen:**
  - **Paar-Programmierung:** aus Programmierer und Inspektor
  - **Interne Begutachtung:**
    - **Inspektion:** Kollege liest Code und prüft vordef. Checkliste. Programmierer erklärt dabei Code, Konventionen und Patterns
    - **Bewertung:** Aller Dokumente + Code von einer anderen Gruppe

- **Mit Tests:**

Statische Analyse	**Dynamische Analyse**
Ohne Ausführung des Programms. Z.B.: Data-Flow Analyse	Mit Tests. Z.B.: Simulation mit konkreten Werten

  - 

    Testprozess: Test nach
  - **Testgetriebene Entwicklung *(iterierend)*:**

1. Schnittstelle einer Methode definieren	2. Testfall für diese erstellen & ausführen	3. Programm-methode schreiben	4. Testfall ausführen, bei Erfolg zum Testsuite hinzufügen

  - **In-Vitro-Test mit Debugger:** Führt Programm aus & kann jederzeit stoppen

- **Breakpoint:** Codezeile, um Ausführung ab zu brechen
- **Watchpoint:** Events, die die Werte einer Variable ändern

▪ **Regressionstest:** Wiederholen von Testfällen,nach Modifizierung v. bereits getesteten Code
  ▪ **Coverage Patterns:** Teste erneut alle Testdaten, risikoreiche Anwendungsfälle, veränderten Code, veränderten Code + deren Abhängigkeiten
  ▪ **In GUI:**
    - **Capture Tools:** Aufnahme der Interaktionen als Skripts
    - **Replay Tools:** Generierung von Ereignissen aus dem Skript *(bspw. Für Testfälle, Wiederholung des Events ohne manuelle Eingaben)*

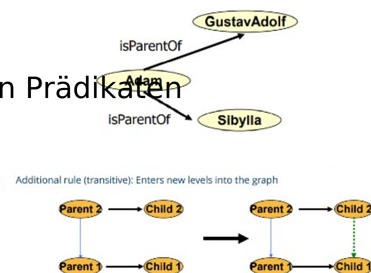
4. Validierung graphenbasierter Modelle & Programme:

- **Graphentypen:**

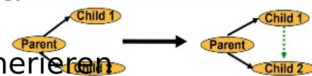
Bäume	**Graphen**	**Listen**
- **Link Tree:** Baum mit Referenzen auf andere Knoten	- **Dag:** Gerichteter azyklischer Graph - **Reduzible:** Zyklen höchst. zw. Geschwister - **Schichtbar mit Skelett-Dags:** Reduzibler, gerichteter Graph mit Referenzen auf andere Knoten - **Unstrukturierter Graph**	

- **Modellvalidierung/Konsistenzüberprüfung:**
  ➢ Beschränkungen als Logik formulierbar & mittels Abfragen auf Graphen anwendbar

- **Analyse von Graphen in Modellen:**
  ➢ **Grundlage: Graph-Logik Isomorphismus:**
    ➢ Logik als Graph, besteht aus Konstanten & binären Prädikaten



  ▪ **Azyklische Graphen layern** ***(schichten)*****:**
    ➢ Mittels **SameGeneration:** fügt Kanten hinzu
    ➢ Anschließend Layers numerieren



  ▪ **Graphen suchen:**
    ➢ Mit SameGeneration und EARS oder DataLog
    - **EARS:** fügt dem Graph Kanten hinzu, die markiert werden können, sodass sie nicht permanent hinzugefügt sind
      Vorteil: Konfluenz = Ergebnis unabhängig der Reihenfolge der Regeln
    - **DataLog:** formale if-then-Regel zum Testen der Prädikate mit Konstanten

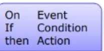
5. Designmethoden *(Entwurfsmethoden)*:

- **Architekturstyle:**
  ➢ Beschreiben grundlegende Organisation und Interaktion zwischen den Komponenten
  ➢ Resultierend aus der Designmethode
  ➢ Bestehen aus Komponenten, Konnektoren, Einschränkungen
  ▪ **Bsp:** MVP, MVC,…

- **Aufrufbasiert:** Komponenten bezeichnen Prozeduren, die sich gegenseitig aufrufen
- **Pipes-and-Streams:** Streams *(Filter)* als Komponenten mittels Pipes *(Channels)* kombiniert, bsp: *cat log.txt|grep error|…*
- **Ereignisbasiert:** Anonyme Kommunikation der Komponenten durch Events
- **Arbeitsflussbasiert:** Workflow beschr. Aktionen bei best. Ereignissen & Bedingungen
- **Repository:** Komponenten sind durch Data-Repositories *(Kapselung d. Objekte der Datenzugriffsschicht, vgl. DB)* verbunden
- **Blackboard:** Aktives Repos., Koordiniert Komponenten durch triggern von Ereignissen

Übersicht der Designmethoden:

- **Funktionorientiert:** Gruppierung Funktionen zu Modulen & Objekten, ohne private Daten
- **Aktionorientiert:** FOD mit Zuständen und Aktionen *(Zustandsbehaftete Funktion)*
  - **Ereignis.-Beding.-Aktionorientiert:** Aktionen werden von Ereignissen geschützt



- **Komponentenorientiert:** Fokus liegt auf wiederverwendbare Komponenten/Modulen
- **Objektorientiert:** Gruppierung von Daten & Aktionen zu Objekten

Design Heuristiken *(Heuristik = „Faustregel")*:

- **Divide & Conquer Strategie:**
  - Probleme in Subprobleme unterteilen *(divide)* und lösen *(conquer)*
  - **Strategien:**

Top-Down	Bottom-Up	Middle-Out
Vom Allgemeinen/ Übergeordneten schrittweise zum Speziellen, Untergeordneten	Gegenteil von Top-Down	Teilprobleme aus Mitte nach Oben werden behoben und durch Verfeinerung gelöst