ソフトウェア工学

2022年11月5日 土曜日

海谷 治彦

目次

- アクティビティ図
- ステートマシン図（状態遷移図）
- データフロー図
- アーキテクチャ設計と配置図
- 品質特性

ユースケース記述を書きましたが

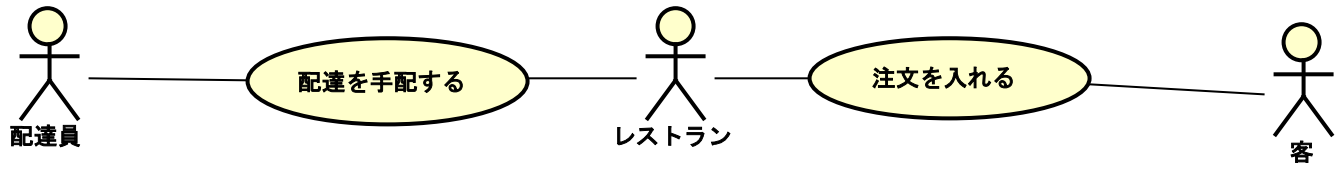
融資する / ユースケース記述

項目	内容
ユースケース	融資する
概要	銀行が融資をする支援をする機能
アクター	顧客 融資担当者 保証人
事前条件	顧客は取引実績がある. 保証人の候補は事前登
事後条件	融資が行われる

基本系列	1 顧客は融資額と保証人候補を入力する. 2 システムは金額確認を融資担当者に送る. 3 システムは保証人候補に許諾確認の連絡を行う. 4 融資担当者は金額を承認する. 5 保証人候補は保証人になることを受託する. 6 システムは顧客に融資をする旨を連絡する. 7 システムは融資担当者に融資が成立したことを連絡する.
代替系列	融資担当者が金額を承認しない場合: システムは顧客に金額再検討を依頼する. 顧客は別の金額を入れ，基本系列3にいく. 保証人候補が受託しなかった場合: システムは顧客に別の保証人候補を入れるように依頼する. 顧客は別の保証人を入れる. 基本系列5から継続
例外系列	融資担当者が金額を承認しない場合，: システムは金額の再検討を顧客に依頼する. 顧客は融資依頼をやめる旨をシステムに入力する. 保証人候補が受託したなかった場合: システムは顧客に別の保証人を入れるように依頼する. 顧客は融資を断念する.

OSの授業スライドより

- ウーバーイーツのシステムの一部.
- 金銭授受関係は省略してある.

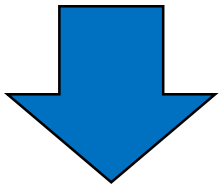


基本系列	1. レストランは配達可能になったことを入力する. 2. システムは配達先に適した配達員に配達依頼をする. 3. 配達員は配達依頼を受諾する. 4. システムはレストランに配達者がとりにくる旨を通知する. 5. システムはレストランに送金する. 6. システムは配達員に送金する.
例外系列	適した配達員がみつからない場合: システムは配達できる人がいない旨をレストランに通知する.

基本系列	1. 客はレストランを選び注文を要求する. 2. システムはレストランに注文を送る. 3. レストランは注文を受諾する. 4. システムは客に注文が受け付けられた旨を連絡する. 5. 客は料金をシステムに送金する.
例外系列	レストランが注文を拒否した場合, システムは注文が拒否された旨を客に連絡する

問題点

- 文系の人には文章なので分かり易いかもだけど，理系にはまどろっこしい．
- 基本的にはアクター群とシステムとのやり取りなのに，そのキャッチボールがわかりにくい．
- 代替系列や例外系列は基本系列の派生なのに，どこで派生するかぱっと見わかりにくい．
- 文章なので，機械的処理には非常に不向き．

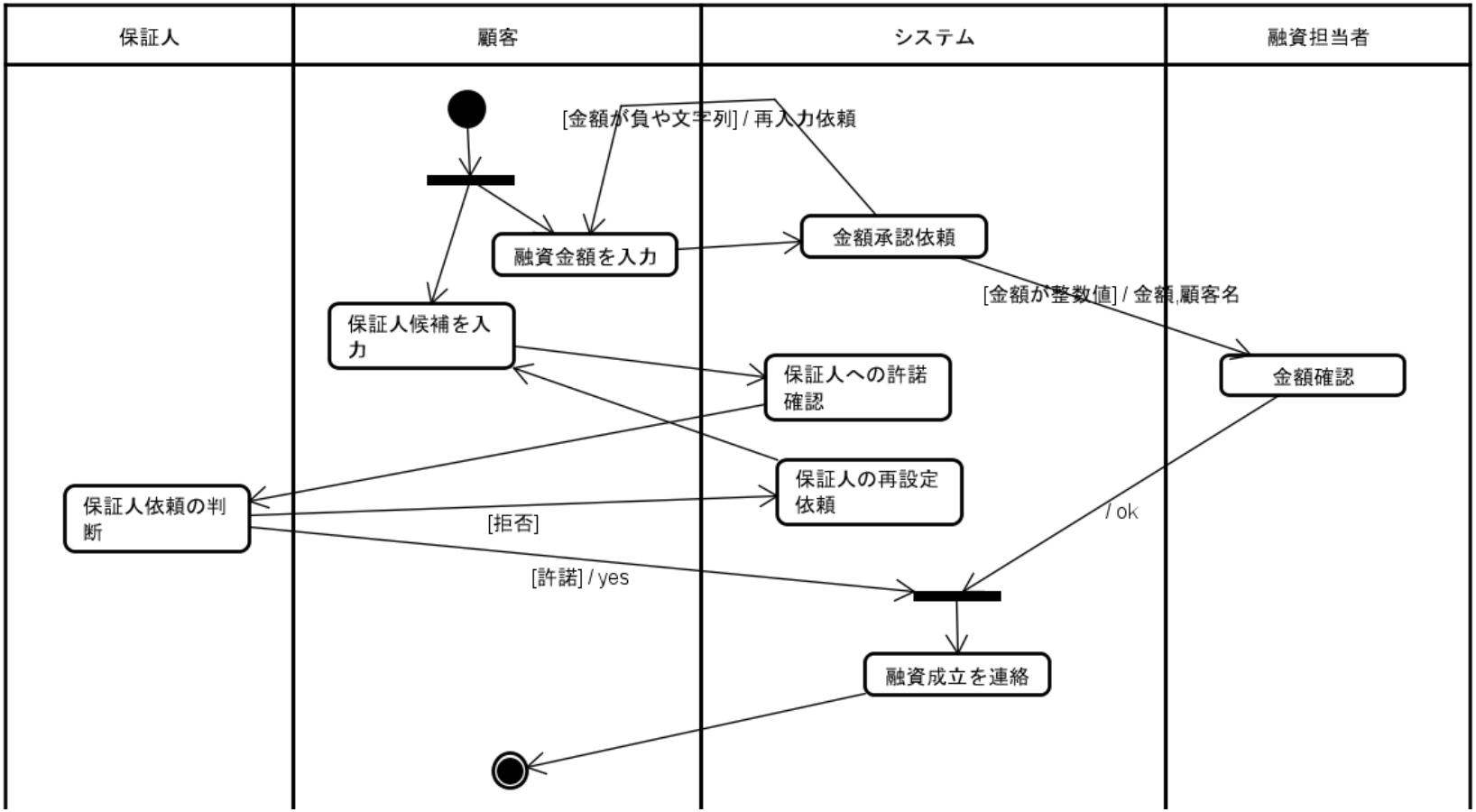


- 図式で書いたほうがいいんじゃない？

アクティビティ図

- UMLの振る舞い的な側面を書く図
  - 他にシーケンス図，ステートマシン図がある.
- 一応，インスタンスの図ではなくクラスレベルの図である.
  - 特定の事例を記述したものではない.
  - 一方，シーケンス図は基本，特定のシーケンス例を書いたインスタンスの図.
- 雰囲気はフローチャートに似ている.
- 理論的な部分としてペトリネットの考えを利用している.
  - 並列に処理してよい流れの見える化
  - 状態遷移図の雰囲気も入っている.
- アクター等の切り分けを明確にできる.
- 所謂，組織のワークフローを書くのに便利.

例 銀行の融資



以降，基本的な文法の紹介.

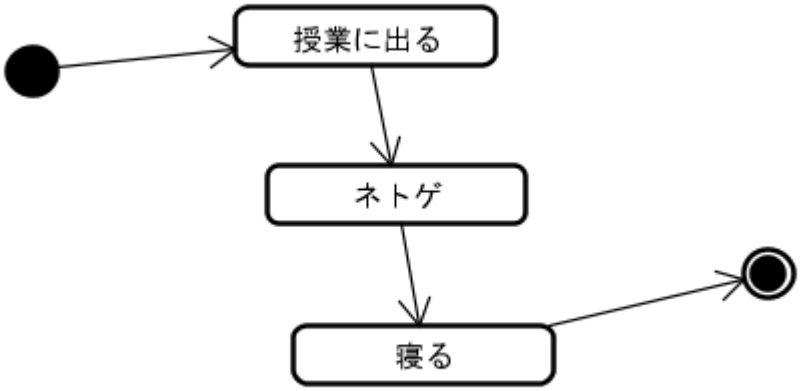
パーティション

- 前頁の外枠それぞれに相当.
- 枠がアクターもしくはシステムに相当する.
- 枠内にあるアクションを枠に対応するアクター等が遂行する.

- 誰が何をやることで，あるユースケースが実施されていくかが俯瞰しやすい.

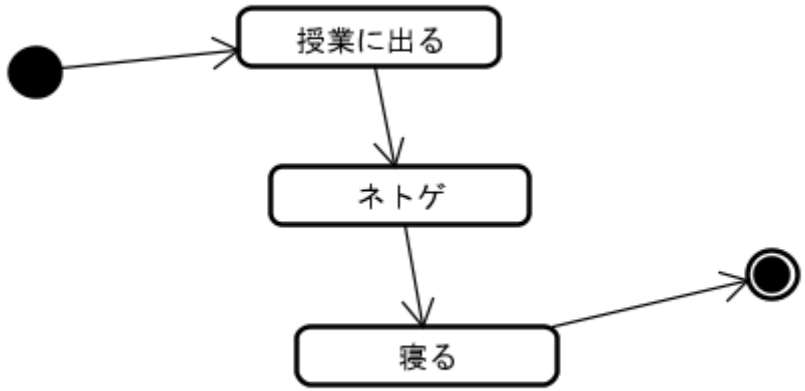
開始と終了ノード

- 一連の処理の開始と終了を示す.
- 終了は複数あってもよい.
- アイコンは状態遷移図やペトリネットに由来する.



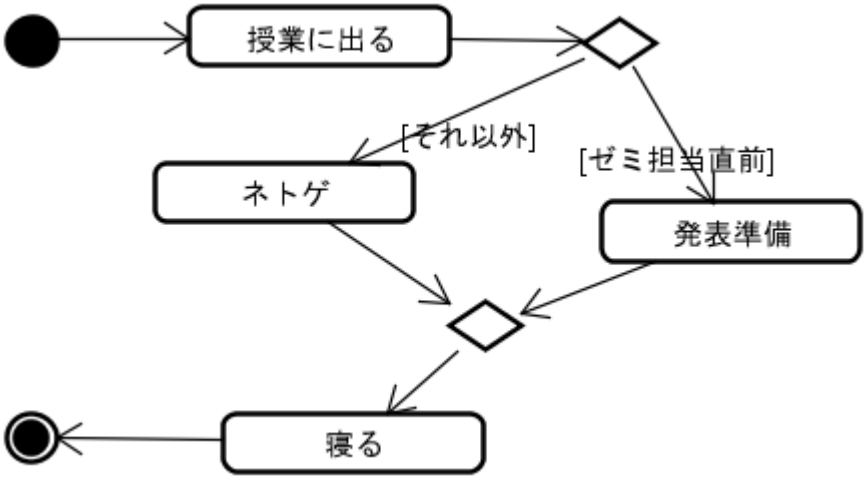
アクションとフロー

- システムもしくはアクターが行うことを示す.
- 「・・・する」に相当.
- フローはアクションの順序を示す.



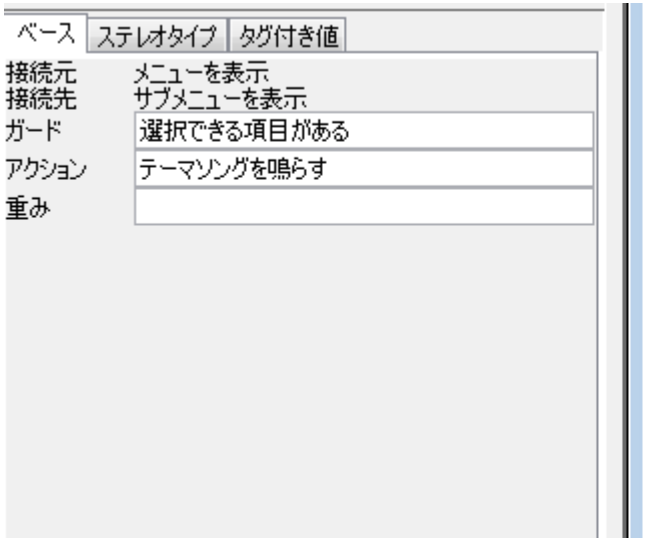
ディシションノード マージノード

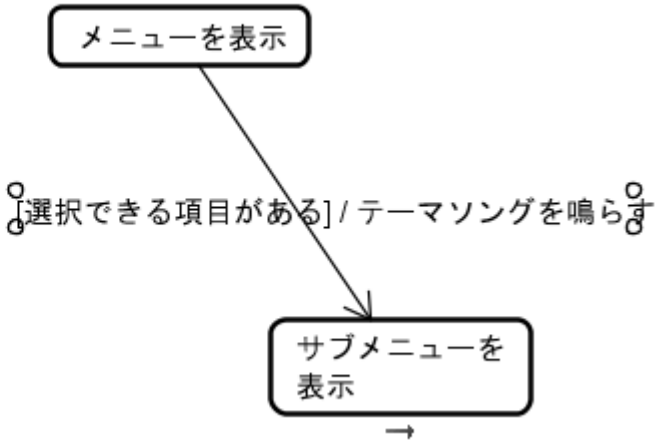
- 条件分岐みたいなのを書ける.
- 分岐は◇で分かれて，◇で合流しないといけない.



ガードとアクション

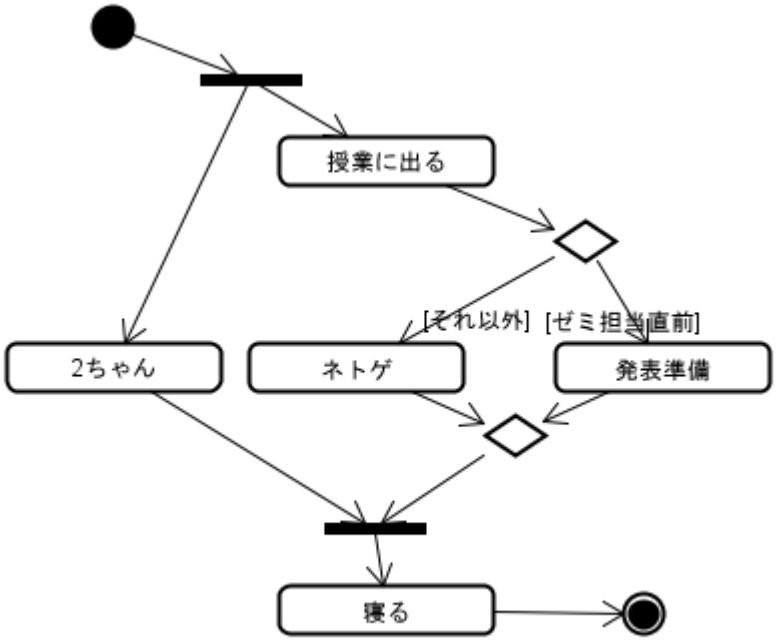
- アクションノード間の遷移において以下の二属性を書ける.
  - ガード その遷移ができる条件
  - アクション 遷移中に起こす行動
- これらの表記は状態遷移図から由来する.





forkとjoin

- 並行処理を書くことができる.
- joinで並行から直列に戻ることが書ける.



書き方のポイント

- あるユースケース(機能)を遂行するのに必要な活動のフローを書く.
- それぞれのアクションを誰がやるのかを識別する.
- パーティションにアクターもしくはシステムを割り当て，それぞれが行うアクションをマッピングする.

問題点

- 複数通りの書き方ができる.
  - プログラム言語でも同様か.
  - 社内，組織内でルールを決める.
- 基本，代替，例外を全部書いたらカオスになって読めない.
  - 色で区別.
  - 予め分けてかく.
  - ツールで分割(というか投射)して読む.
- 制御フローとデータフローがごっちゃになる.
  - 今回は意図的にデータフロー部分は解説しませんでした.
- 文系人が嫌がるかも(特に老人).
  - やつらに勉強させる.

ステートマシン図

- 状態遷移図のUML版
- かなり複雑なことがかける.

- OSの授業とかぶる・・・

状態とコンピュータ

- 人間でさえ，自身の状態を「起きてる」，「寝てる」等，変えてゆく.
- 同様にコンピュータやソフトウェアは，その状態を変化させてゆくことで処理を行っている.
- このような状態変化をモデル化するのが状態遷移図であり，そのUMLでの名前がステートマシン図である.

- コンピュータ分野でポピュラーな分野である「オートマトン」も状態遷移図とほぼ同じと考えてよい.

状態と処理

- フローチャートやアクティビティ図では処理がノード(丸や矩形)として記述される.
- 状態遷移図は，これと逆で，処理が線となる.
- 見た目がアクティビティ図と似てるが，この点が大きく違う.

- この辺の感覚に慣れないと状態遷移図で処理っぽいものをノードとしてしまう場合があるので注意.

OSより

UIのためのステートマシン図

- 本来は，システム内のオブジェクトの状態をモデル化する図.
- 前述のユースケース記述を画面遷移として記述するための記法と考えてよい.

■角の丸い四角 = 画面

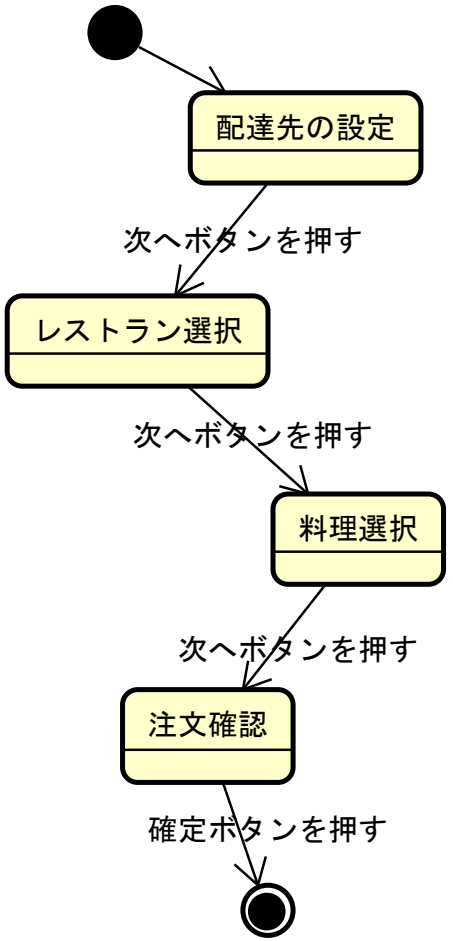
■矢印 = 画面遷移

と考えてよい.（若干，おおざっぱ）

- 丸が処理であるフローチャート等とは考え方が逆.
  - 状態にはある程度の時間幅があるものと考える.

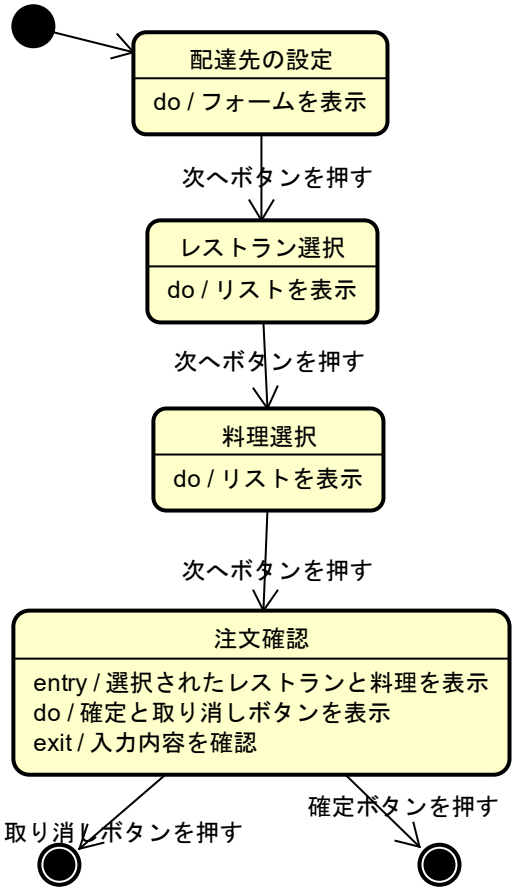
簡単な例 出前の注文

- 状態が画面で，遷移がボタンを押す等のアクションである典型例.
- レストラン，料理の選択は逆の仕様もありうる.
- 配達先は最初に指定しないと，レストランは絞り込めない.
- 状態の名前に「～画面」とつけると意味がしっくりくる.



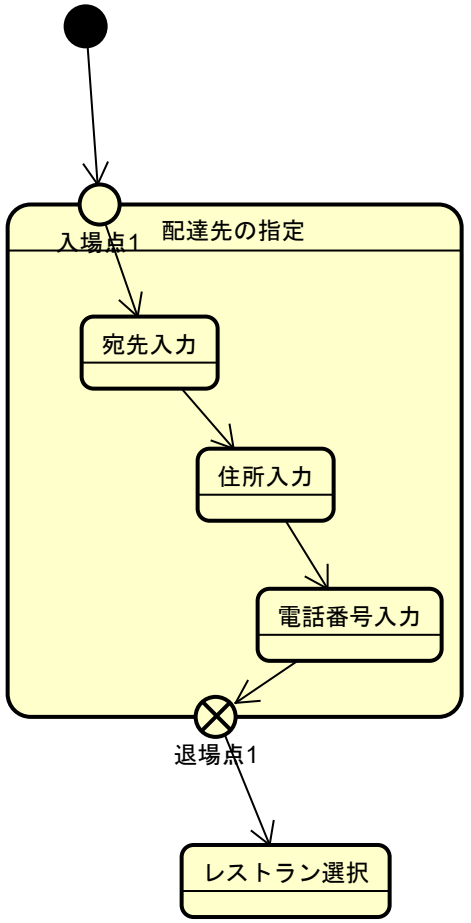
状態中の動作を指定できる

- 状態にかかわる動作を指定できる.
- entry
  - 状態に遷移した際に行う動作
- do
  - 状態中に行う動作
- exit
  - 他の状態等に遷移してしまう時点で行う動作



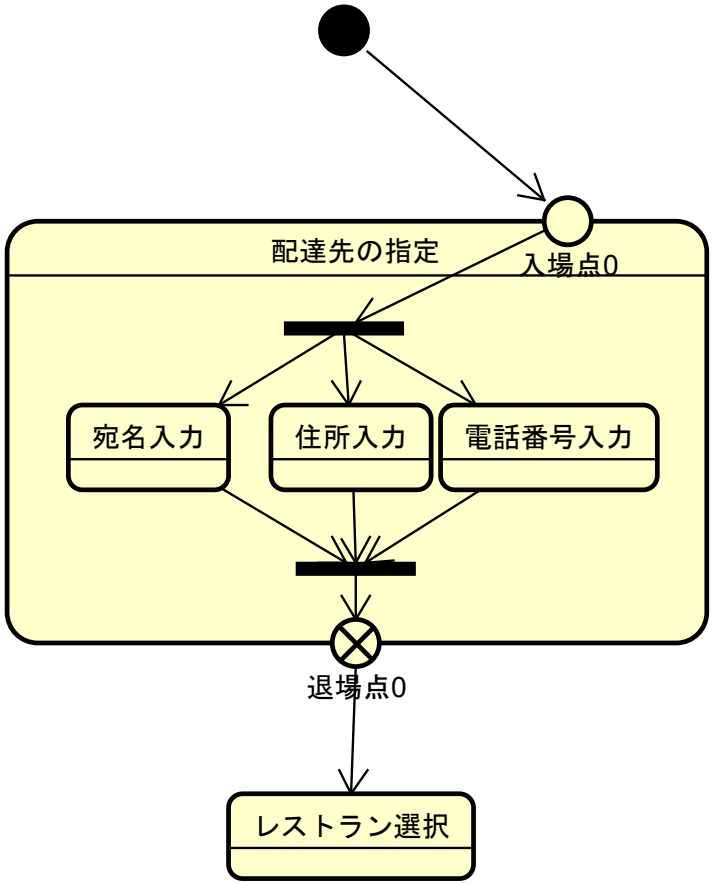
部分状態

- 配達先の指定といっても，名前入力状態，住所入力状態，電話番号入力状態と，より細かい状態があると考えてもよい．
- そのような状態を，状態の中に書いてもよい．
- まぁ，どの状態をグループ化するかは意味次第．



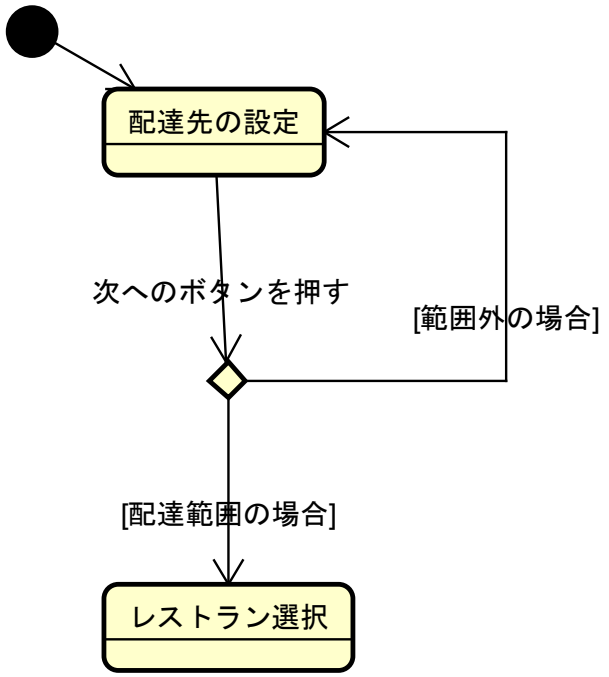
同時並行の状態

- フォーム等に，氏名，住所，電話番号等を任意の順番に埋める場合を考える.
- それぞれに，入力済/未という状態を持つととらえることができる.
- また，入力の順番も特に問わないので，並行して確定してよいということになる.
- そのような場合は右のように書いてもよい.
- まぁ，ここまで詳しく書く必要は無いだろう.



状態の値で条件分岐

- ボタンを押す等の動作ではなく，状態が保持する値によって，遷移先を変えることも可能.
- プログラムの if then とほぼ同じ.

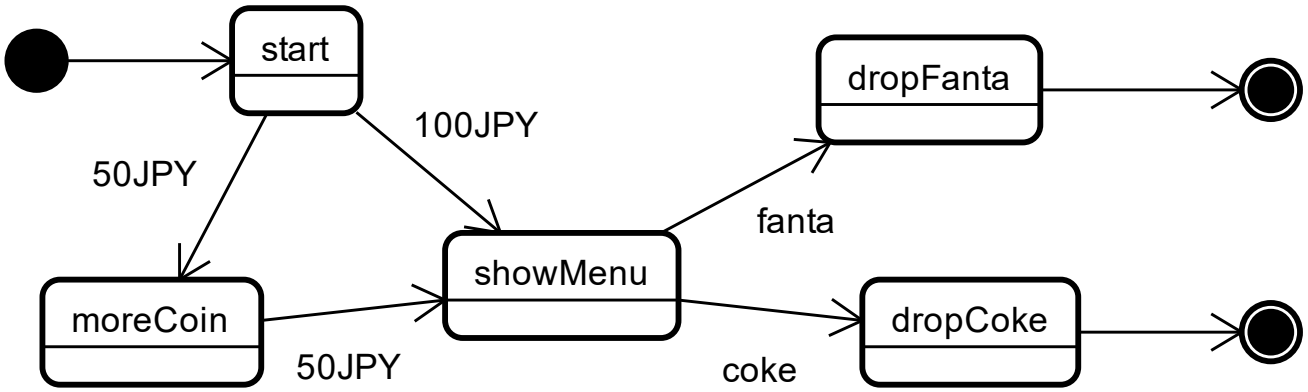


システム状態のモデル化

- 本来的には，こちらがステートマシン図の一般的な使い方．
- 最小粒度では，クラス図のクラス毎にステートマシンを書いてよい．
  - 後のほうのじゃんけんPlayerの例
- 現実的には，ある程度のクラスの組み合わせをモデル化する．

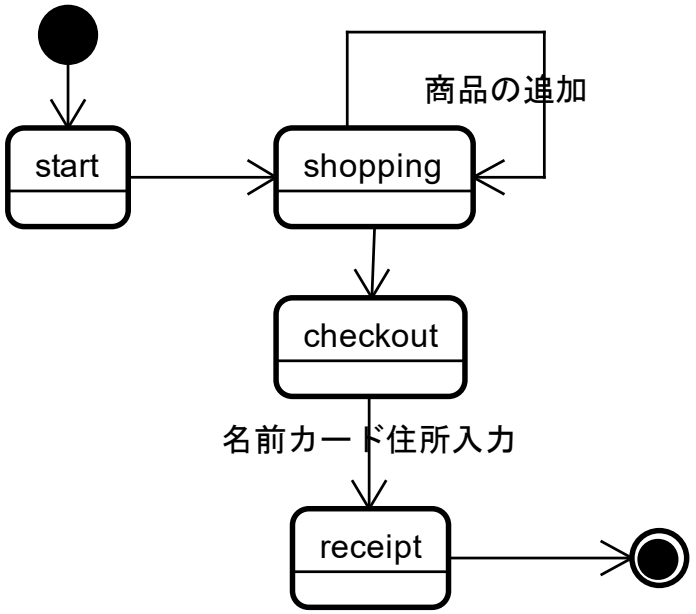
簡単な自動販売機

- 100円でファンタもしくはコーラが買える.
- 100円玉と50円玉しか使えない.
- 100円分お金をいれるとメニューが表示される.



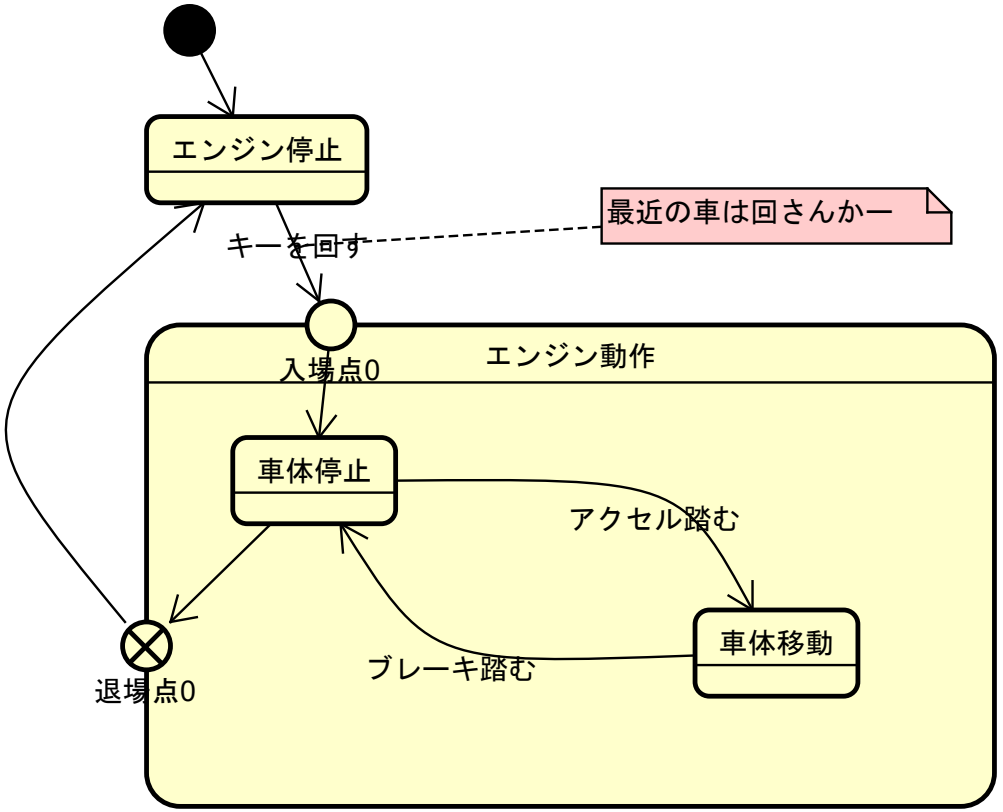
ショッピングサイト

- 簡単なオンライン買い物サイト
  - ビール限定
- ちゃんと購入物の累積情報がページ間で継承されている.

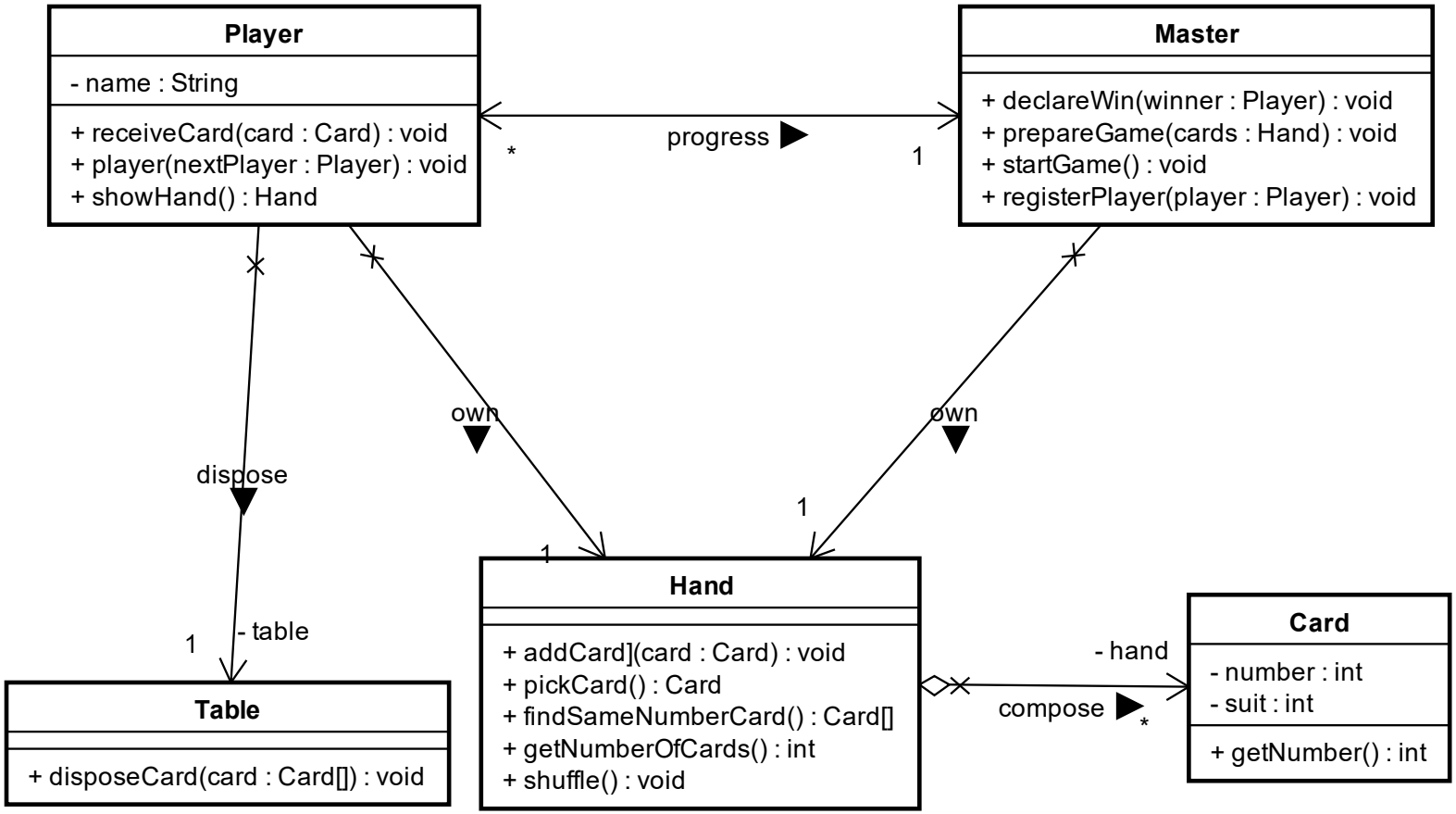


自動車の状態

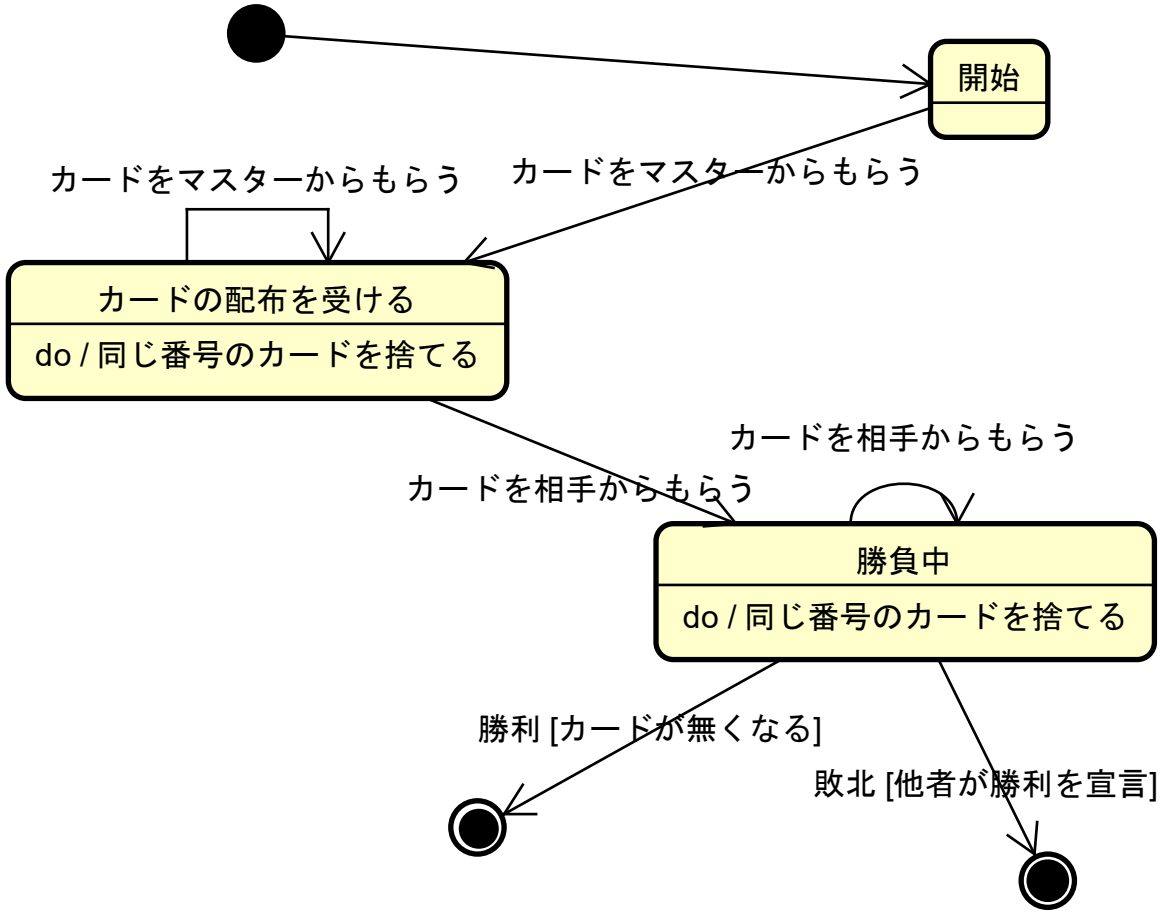
- 以下の二つの状態変数があるが，独立はしてない．
- エンジン停止/稼働
- 車体停止/移動



ばば抜きゲームのクラス図



Playerの状態モデル



データフロー図

- 通称，DFD
  - Data Flow Diagram
- 処理と入出力データの関係のモデル.
- C言語等の手続き型言語との親和性が高い.

- UMLの一部ではない.
- 年配(50台以上)のSEなら，誰でも知っている.
- astah で記述できる.

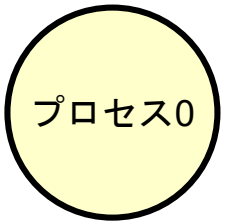
その他，特徴

- 構造化設計(C言語等の設計)を素直に表現できる.
- UML登場前は広く使われた.
- データの流れを書くので，セキュリティ界隈では，わりと今でも重宝される.
  - マイクロソフトの STRIDE 等.
- UNIX/Linuxのパイプ and フィルタの考え方も素直にかける.

- UMLにおけるユースケース図にほぼ対応する.

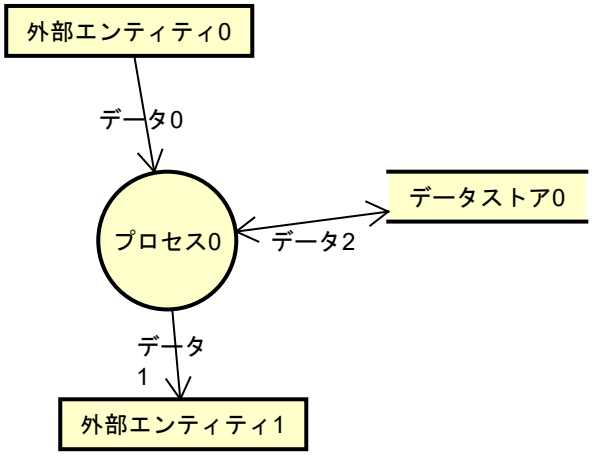
記法

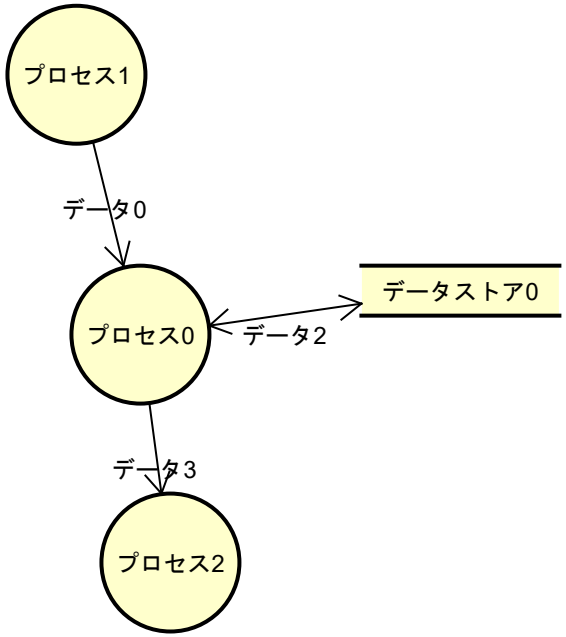
- プロセス
  - 処理を表す.
  - 関数とほぼ対応する概念.
- データフロー
  - データの入出力を示す線.
  - 関数の引数や返値に対応
- エンティティ
  - システム外部とのインタフェース
- データストア
  - データを保持するデータベース的なもの.
  - 画としてはコンデンサ(電子部品)のアイコン.



基本的な書き方

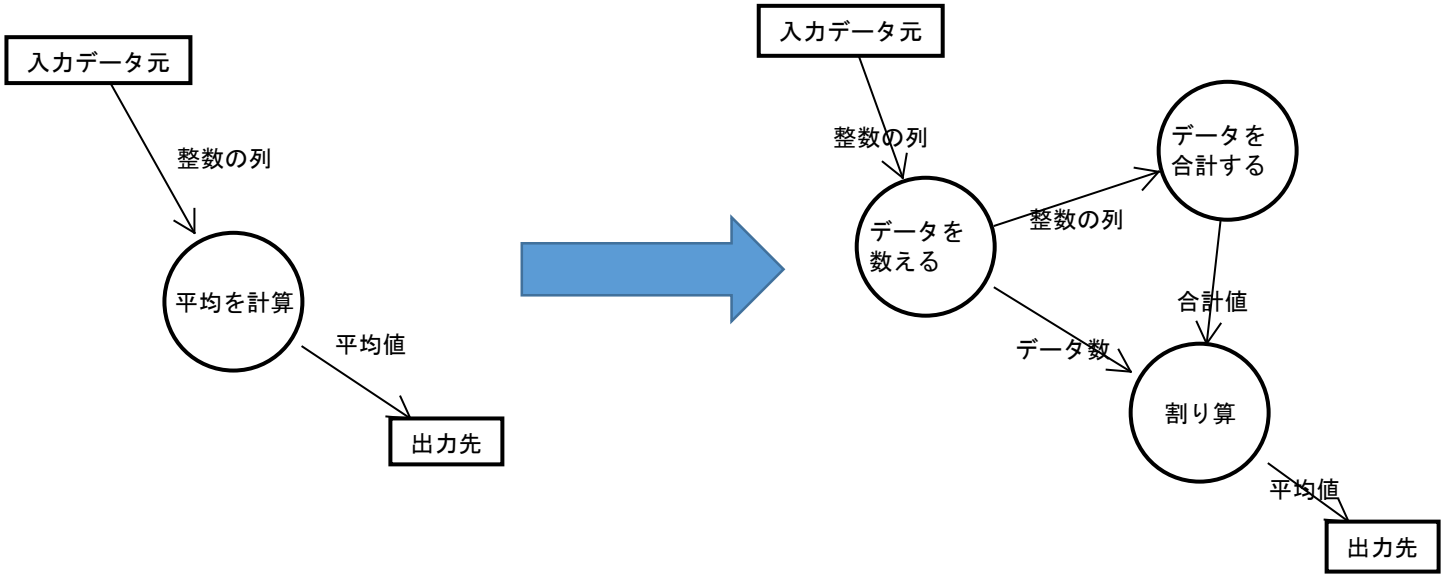
- 外部エンティティや他のプロセスからデータを受け取る.
  - 入出力は複数あってもよい.
- なんか計算をプロセスでして，データを吐き出す.
- 必要に応じてデータストアの値を参照してよい.
  - 無くても良い.





階層的な記述

- 複雑なプロセスは単純な複数のプロセスに分割してよい.
- 完全にC言語の関数分割の考え方と同じ.



ちょっと有名な例: 酒屋の在庫問題

ある酒類販売会社の倉庫では，毎日数個のコンテナが搬入されてくる．その内容はビン詰めの酒で，1 つのコンテナには10 銘柄まで混載できる．扱い銘柄は約200 種類ある．倉庫係は，コンテナを受け取りそのまま倉庫に保管し，積荷票を受付係へ手渡す．また受付係からの出庫指示によって内蔵品を出 庫することになっている．内蔵品は別のコンテナに詰め替えたり，別の場所に保管することはない．

空になったコンテナはすぐに搬出される．

積荷票: コンテナ番号(5 桁) 搬入年月，日時 内蔵品名，数量(の繰り返し)

さて受付係は毎日数十件の出庫依頼を受け，その都度倉庫係へ出庫指示書を出すことになっている．出庫依頼は出庫依頼票または電話によるものとし，1 件の依頼では，1 銘柄のみに限られている．在庫がないか数量が不足の場合には，その旨依頼者に電話連絡し，同時に在庫不足リストに記入する．そして当該品の積荷が必要量あった時点で，不足品の出庫指示をする．また空になるコンテナを倉庫係に 知らせることになっている．

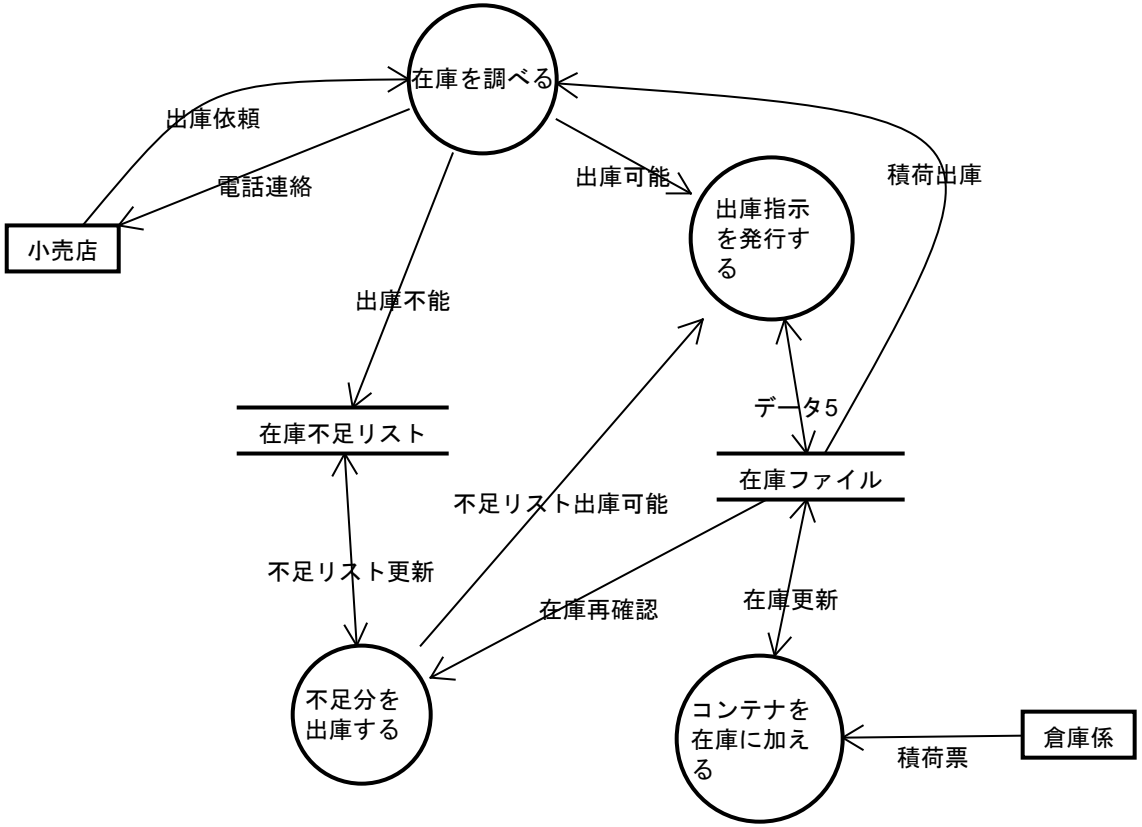
出庫依頼: 品名，数量 送り先名

受付係の仕事(在庫なし連絡，出庫指示書作成および在庫不足リスト作成) のための計算機プログラム を作成せよ．

出庫指示書: 注文番号 送り先名
コンテナ番号 品名，数量 空コンテナ搬出マ?ク (の繰返し)
在庫不足リスト: 送り先名 品名，数量

なお移送や倉庫保管中に酒類の損失は生じない．
この課題は現実的でない部分もあるので，入力データのエラー処理などは簡略に扱ってよい．
以上あいまいな点は，適当に解釈して下さい．

データフロー図のモデル



アーキテクチャとは？

- Architecture もともとは建築用語
- コンピュータでは CPUやハードの構造のことを指すほうが普通.
- システム アーキテクチャ，ソフトウェア アーキテクチャとも呼ばれる使い方もされるようになった.
- 以下を図式で書くことを指す場合が多い.
  - システム内のソフトウェア部品間の論理的な構造
  - システムが動くハードウェアと通信路上のソフトウェア部品の配置位置.

- レスポンスの速さ，処理能力，信頼性等のシステムの品質はアーキテクチャに依存する場合が多い.

ソフトウェアの部品

- 昨今のソフトウェアは複数のプログラムで構成される場合が多い.
  - 例 ウエブサーバー，データベースサーバー，ウエブブラウザ(UI担当)
- また，既存のライブラリやフレームワーク(後述)を使うことも多い.
  - 例 暗号関係部品(openssl等)，UIフレームワーク(Bootstrap等)，アプリケーションフレームワーク(cakePHPやPlay等)
- 上記の意味からプログラム，ライブラリ，フレームワーク等をソフトウェアの部品と呼ぶ.

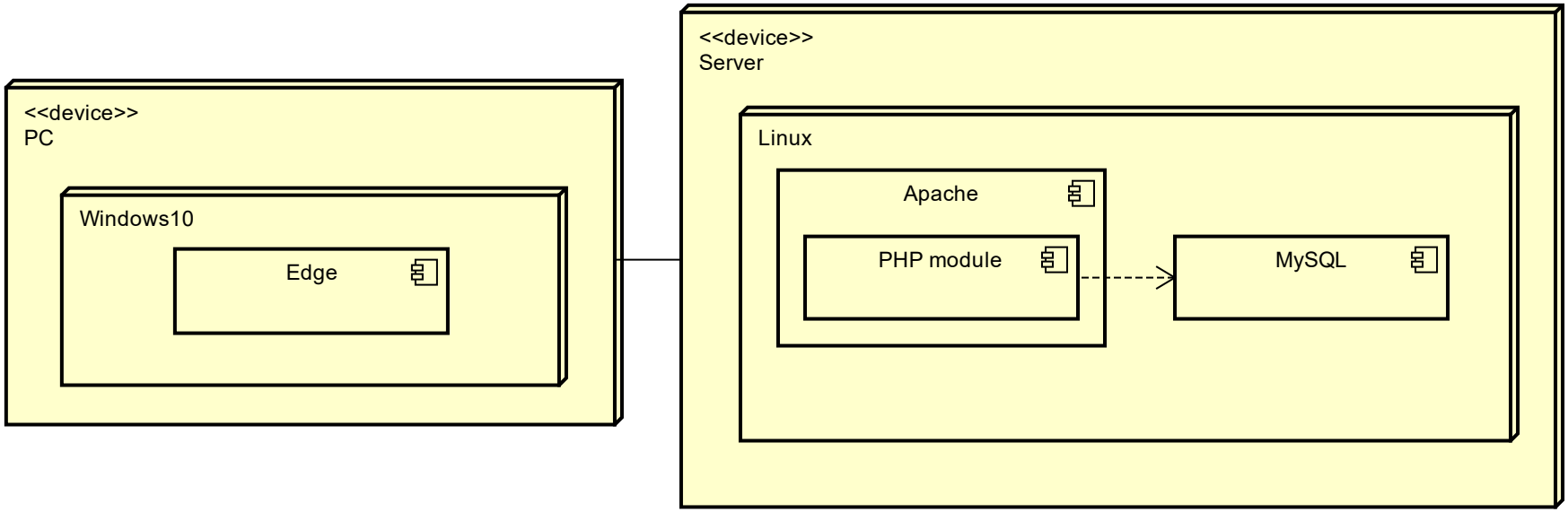
配置図

- アーキテチャを書くのに便利な図
- UMLの一部である.
- 無論，astah でもかける.

astahで配置図を書く場合

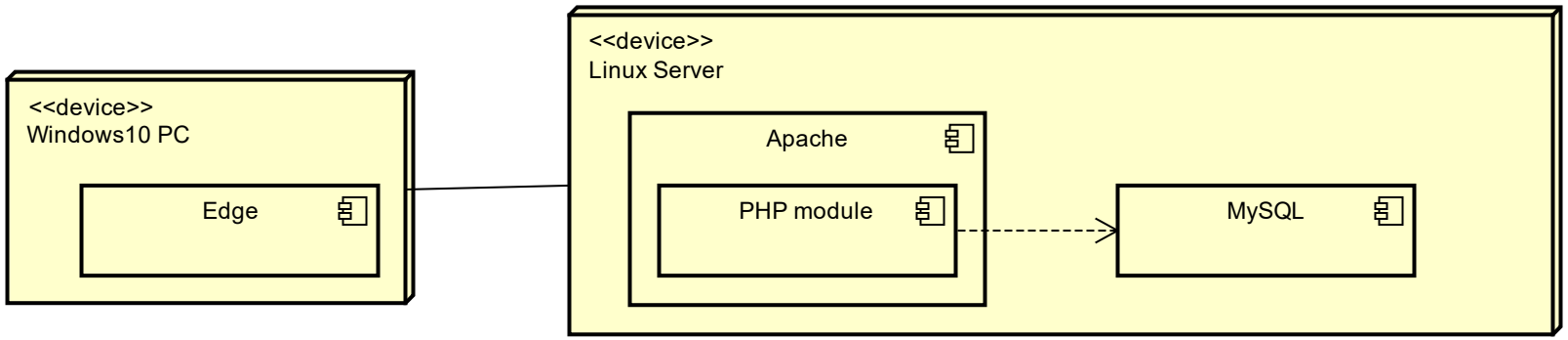
- node と component を使ってください.
- node instance と component instance は使わないでください.
- nodeの例
  - PC等のハードウェア <<device>> をつける
  - OS 等の実行環境
  - 仮想マシンを動かす環境 VMware 等
- componentの例
  - 実行可能プログラム，データベース等を含む.
  - (共通)ライブラリ
  - 設定ファイル
- 関連
  - 通信
- 依存
  - componentにおいて，使うほうから使われる方へ

簡単なWeb application の例



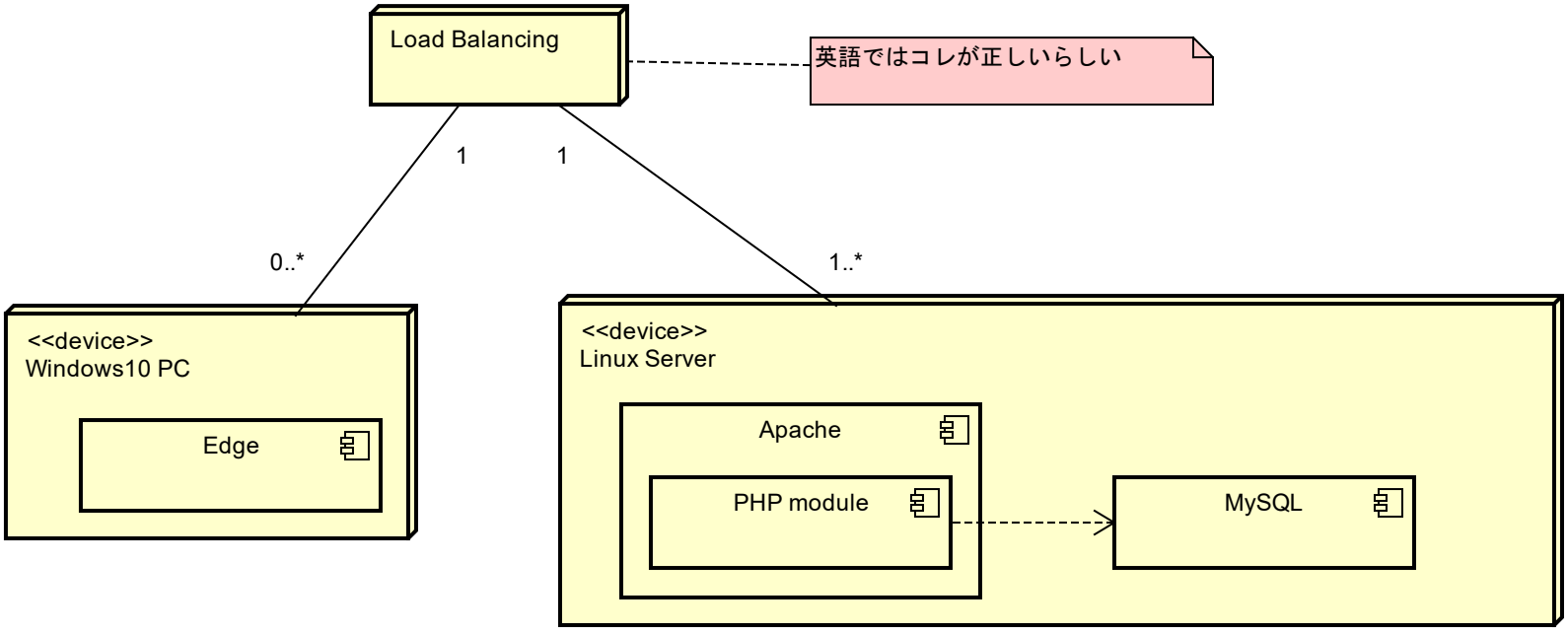
deviceとOSは一体でもいいかも

- 厳密には前ページだが，一つのdeviceに一つのOSが普通なので.
- 尚，仮想マシン等を使うと話が異なる.



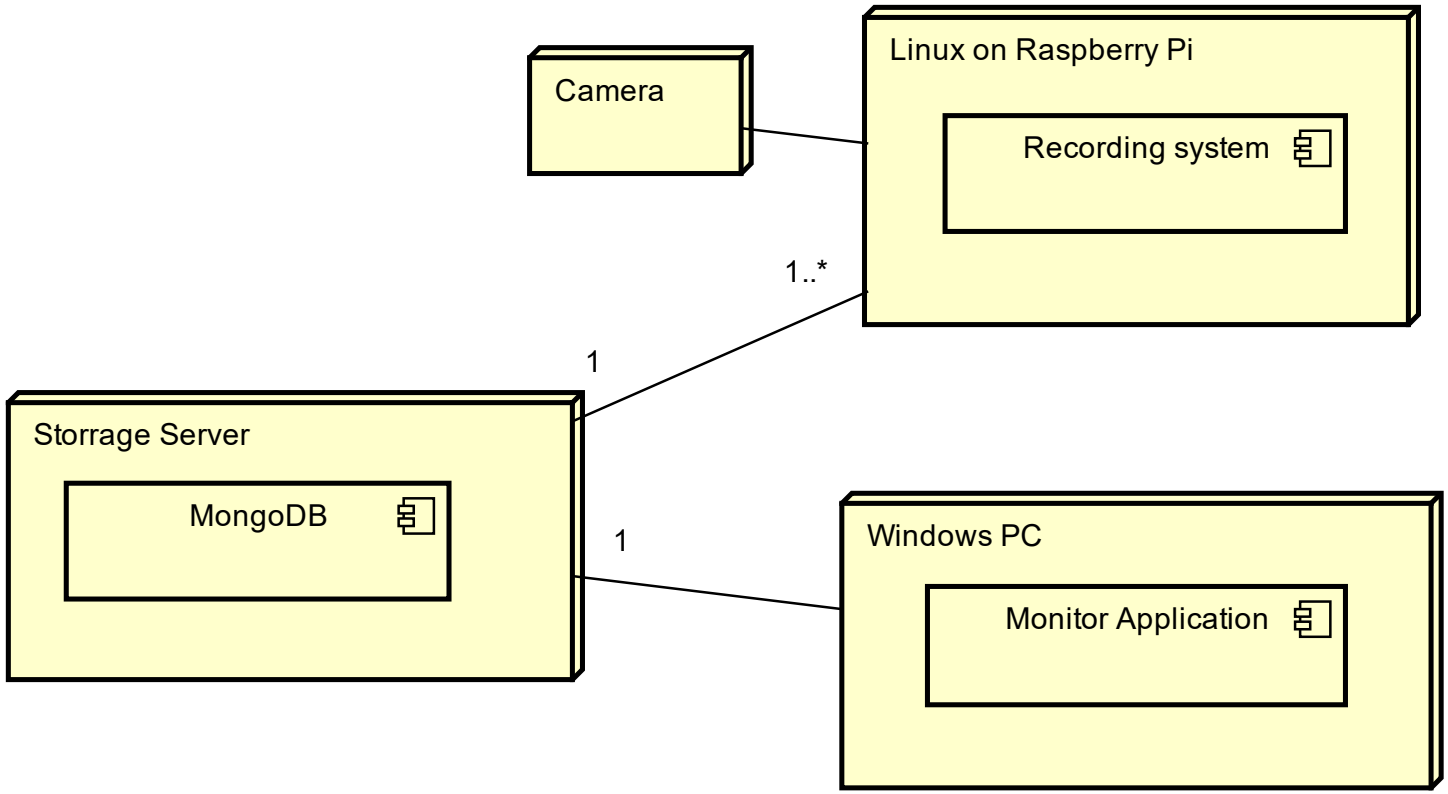
ロードバランサー

- 過負荷に耐えるように装置を多重化する仕組み.
- 一斉アクセスがあるサイトはコレな無いと死ぬ.



監視カメラシステム

- 建物とかの監視カメラシステム
- 多少，fog や edge computing っぽいかんじ.



配置図を書く事で

- どんなハードを準備する必要があるか認識
- ハード間の連携を明確化
- それぞれのハードにどんなソフトを置くかを決める

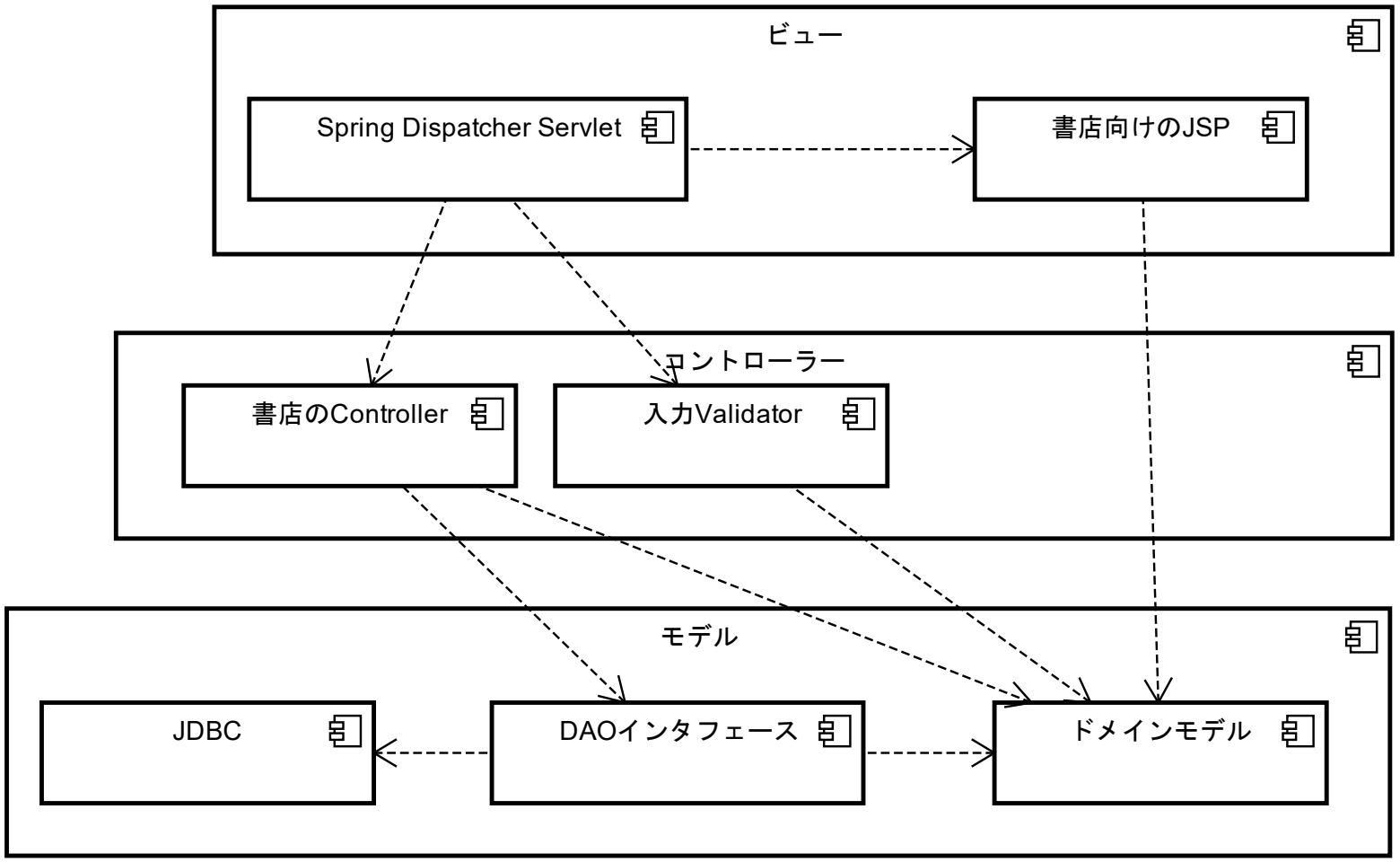
- それぞれのコンポーネントは，クラス図等で構造を明確化する.
  - 新規に開発するのであれば.

ソフトウェア部品の依存関係

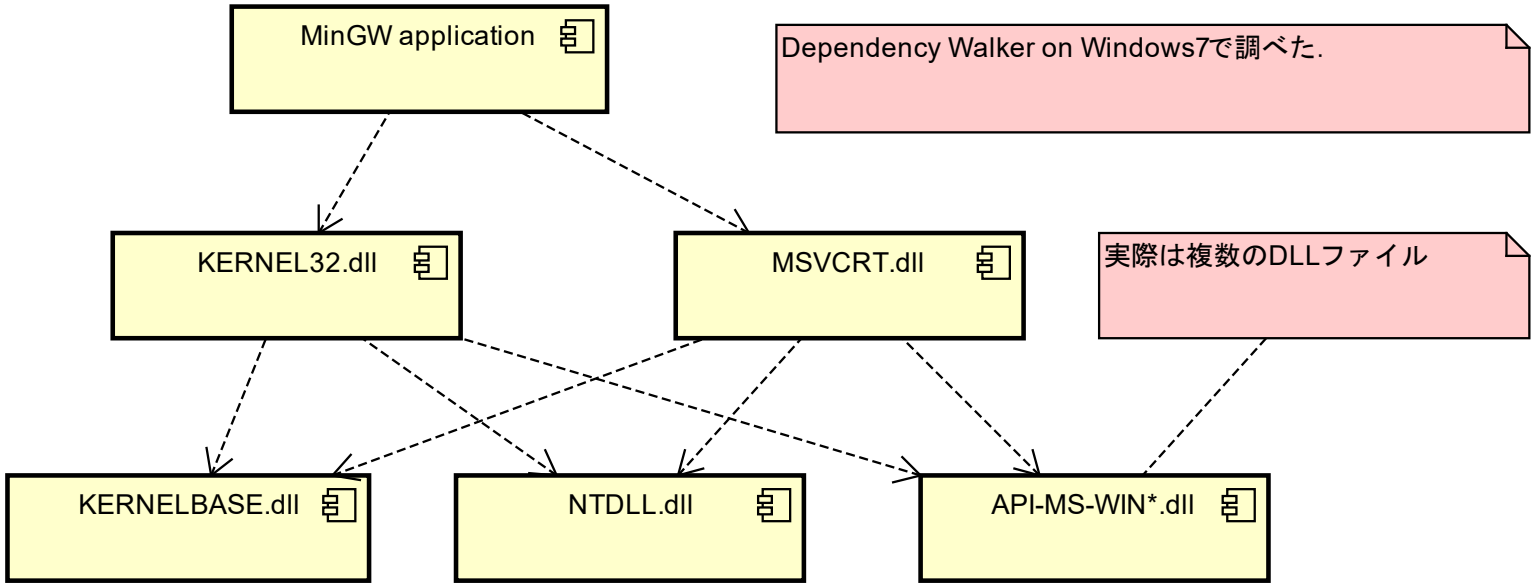
- デバイス等のnodeは気にしないで，componentの依存関係を書いておくこともできる.
- 本来は，コンポーネント図で書くのだが，まぁ，配置図でも当面はよいであろう.

- 設定ファイルやライブラリ等のこまごまとしたソフトウェア部品の関係を書くのに便利.

MVCと該当するコンポーネント群



MinGWで作った Windowsで動くプログラム



品質特性

- ソフトウェアや情報システムは，それらが提供する機能に加え，期待される品質特性がある．
  - 例えば，銀行等の入出金機能は，「正確」かつ「高信頼性」(故障が無い)であることが期待される．
  - スマフォ等のソフトでは，例えば電池の持ちが良い等の「パフォーマンス」も期待される．
- これら品質特性は，往々にして，設計やアーキテクチャ決定に大きく依存する．
- パターンというのは，広い意味で，ある品質特性を担保しやすくする設計，実装の定石ともいえる．
  - 前回，紹介したデザインパターンは，「拡張性」という品質に注目しているものが多い．

品質特性とは？

- いくつかの異なる標準規格や，著書等で，いくつかの品質特性のカタログが提示されている.
- しかし，そのほとんどは大きな違いは無い.

- ISO9126
- SQuaRE
- IEEE830
- FURPS+
- NFR framework

ソフトウェア品質特性標準 – ISO 9126

- **ソフトウェア品質特性**に関する標準
- http://www.cam.hi-ho.ne.jp/adamosute/software-quality/iso9126.htm より

品質特性(quality charastristics)	品質副特性(quality subcharactristics)	主な内容
機能性(fuctionality)	合目的性(suitablity) 正確性(accuracy) 相互運用性(interoperability) 標準適合性(compliance) セキュリティ(security)	目的から求められる必要な機能の実装の度合い
信頼性(reliability)	成熟性(maturity) 障害許容性(fault tolerance) 回復性(recoverability)	機能が正常動作し続ける度合い
使用性	理解性 習得性 運用性	分かりやすさ、使いやすさの度合い
効率性	時間効率性 資源効率性	目的達成のために使用する資源の度合い
保守性	解析性 変更性 安定性 試験性	保守(改訂)作業に必要な努力の度合い
移植性	環境適用性 設置性 規格適合性 置換性	別環境へ移した際そのまま動作する度合い

ソフトウェア品質特性標準 – IEEE 830-1998

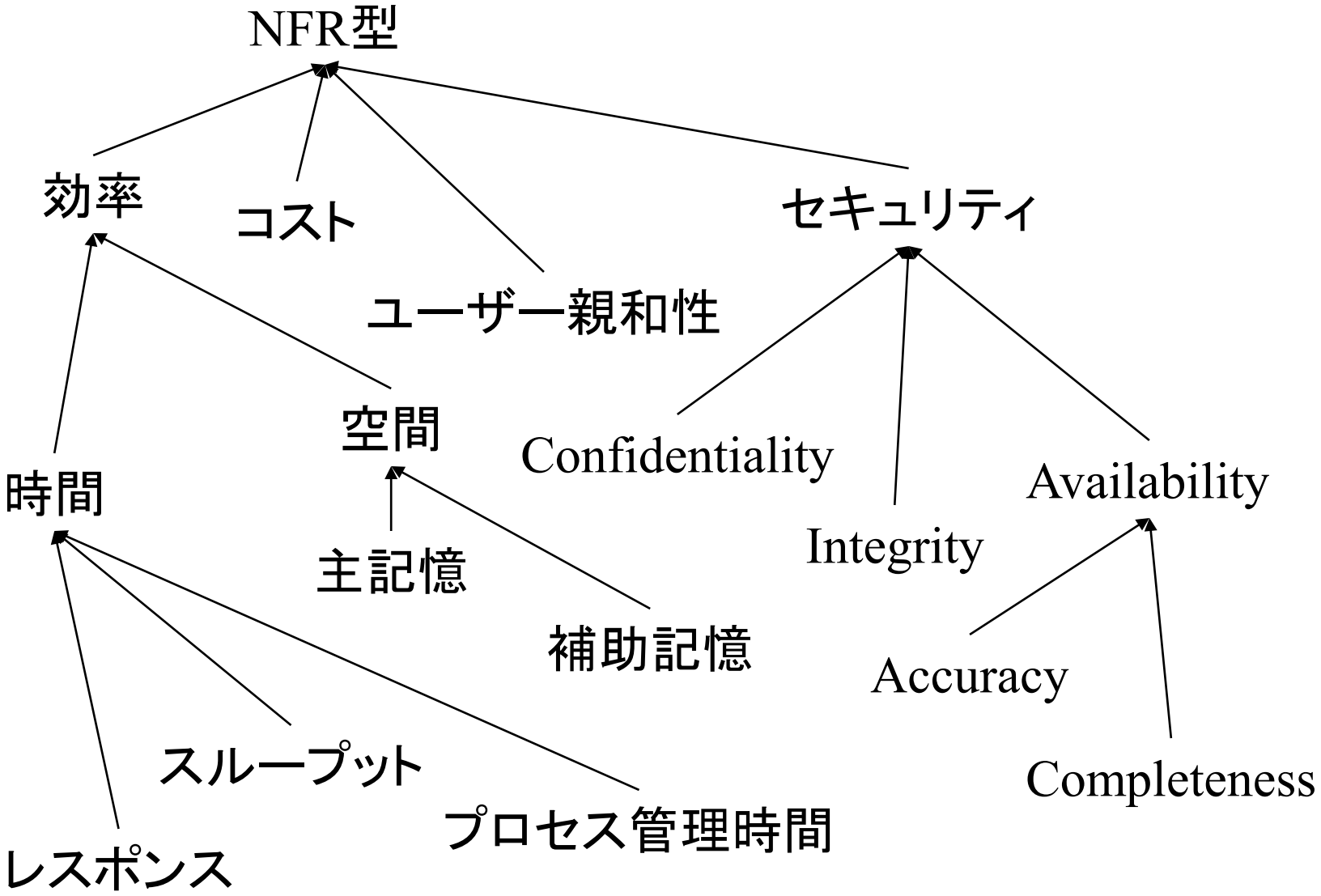
- ソフトウェア要求仕様に対する推奨プラクティス
- 「3.6 Software system attributes」にて、ソフトウェア品質特性を規定
  - 信頼性(Reliability)
  - 可用性(Availability)
  - セキュリティ(Security)
  - 保守性(Maintainability)
  - 移植性(Portability)

分析観点の例: FURPS+モデル

- Grady, “Practical Software Metrics for Project Management and Process Improvement” (Prentice-Hall, 1992)
- http://www.atmarkit.co.jp/farc/rensai/re_mgt02/re_mgt02.html より

機能 (Functionality)	機能	セキュリティ	機能要求
使いやすさ (Usability)	使いやすさ 美しさ	覚えやすさ ユーザー向け文書	機能外要求
信頼性 (Reliability)	故障の頻度／深刻さ 復旧のしやすさ	予測のしやすさ 精度 平均故障間隔	
性能 (Performance)	処理速度 メモリ使用量	スループット 応答時間	
サポートのしやすさ (Supportability)	テスト性 拡張性 適応性 保守性 互換性	構成容易性 保守容易性 インストール容易性 ローカライズ容易性 頑強性	

NFR frameworkによる階層分類



品質特性を決めたらどうするか？

- 信頼性や効率性等の特性は，そのままでは実現することができない．
- 通常は，以下のどちらかに翻訳して実現する．

1. 特性をアーキテクチャに翻訳する．信頼性等はこちらが多い．
2. 特性を機能に翻訳する．セキュリティやユーザビリティは大体こっち．

- 本来，顧客等が必要としていない機能（ユーザー認証等）が結局含まれるのは，これら特性が別途，非機能要求として望まれるため．

本日は以上