# Software Design
# 8. Design and Modeling

Natsuko Noda
nnoda@shibaura-it.ac.jp




# ソフトウェア設計論
# 8. 設計とモデリング

野田 夏子
nnoda@shibaura-it.ac.jp




## Today's Topics

- Software design using models
  - Relationship between diagram, model, and design
- Module & Modularization
- Design viewpoints




## 本日のお題

- モデルを使ったソフトウェア設計
  - 図、モデル、設計の関係
- モジュールとモジュール化
- 設計の視点

# Software Design




# ソフトウェア設計




## What we learnt

- We learnt UML diagrams.
- Static modeling
  - Definition :
    Class diagram
  - Example :
    Object diagram
- Dynamic modeling
  - Definition :
    State machine diagram
  - Example :
    Sequence diagram, communication diagram




## これまでに学んだこと

- UMLの図
- 静的モデリング
  - 定義：
    クラス図
  - 例示：
    オブジェクト図
- 動的モデリング
  - 定義：
    ステートマシン図
  - 例示：
    シーケンス図, コミュニケーション図

# What we learnt

- These diagrams can be used to describe different levels of things.
  - For example, class diagram:
    - can describe "concepts" of our real world.
      - e.g. "Students" belong to a "University", and "University" has many "Departments", and …
    - can describe the structure of programs.
      - e.g., a class structure of a Java program.
- We learnt notations of UML diagrams.
- We learnt essentials of modeling.
- We learnt tools for designing software.
  - Not designing software itself.




# これまでに学んだこと

- これらの図は様々なレベルのことを記述するために使える
  - 例えば、クラス図なら:
    - 実世界の概念を表現できる
      - 例. "学生" は "大学" に所属，"大学" には多くの "学部" がある，…
    - プログラムの構造を表現できる
      - 例. Javaプログラムのクラス構造
- UML図を学んだ
- モデリングの本質を学んだ
- ソフトウェア設計のためのツールを学んだ
  - ソフトウェア設計そのものではない




# Software design

- An activity to decide software structure, necessary components, their relationship, and their properties.
- A result of above activity.
- Software design is based on software requirements; the design must satisfy the requirements.
- The design is translated into an executable program. → implementation




# ソフトウェア設計

- ソフトウェア構造、構成要素、構成要素間の関係、およびそれらの特性を決める作業
- 上記作業の結果
- ソフトウェア設計は、ソフトウェアの要求に基づく。つまりソフトウェア設計は要求を満たさなければならない
- ソフトウェア設計は、実行可能なプログラムに変換される → 実装(implementation)という

cf. SE-J, 5.1.1

# Design activities

- External design; decision of external interface
  - to determine the interface by which the software interacts with the outside of the system.
    - man-machine interface
    - format of forms
    - communication interface
- Internal design; decision of internal structure
  - to define internal components and the relationships between them, and also to decide the detail of each component.
    - Including architecture design; design basic structure
  - modularization




# 設計に関わる作業

- 外部設計; 外部のインタフェースの決定
  - ソフトウェアがその外部とどのようなインタフェースを通してやりとりするのかを決定する
    - マンマシンインタフェース
    - 帳票などの形式
    - 通信のインタフェース
- 内部設計; 内部の実現構造の決定
  - ソフトウェアの実現方式、すなわち内部の構成要素とその間の関係を定義するとともに、個々の構成要素の実現の詳細を決める
    - アーキテクチャ(基本的な構造)の設計を含む
  - モジュール化

cf. SE-J, 5.1.2




# Design activities (cont.)

- Decision on realization technologies
  - Decision on hardware, OS, middleware, etc.
  - Decision on technologies we use and/or standards we comply with.
- Verification of design
  - Check whether the design satisfies the requirements or not.




# 設計に関わる作業 (cont.)

- 実現技術の決定
  - どのようなハードウェア、OS、ミドルウェアなど利用するか
  - どのような標準に準拠するか
- 設計の確認
  - 設計が要求を満たすものであるかどうかを確認

# Requirements definition and design

- Requirements definition – define "what"
- Design – define "how"
- Requirements specification is an input to design.
  - Basically, first define requirements, and then design.
  - But both are proceeded back and forth; define requirements → design → define more detailed requirements → detailed design → …




# 要求定義と設計

- 要求定義 – "what"を決める
- 設計 – "how"を定義する
- 要求仕様は設計へのインプットになる
  - 基本的には、要求を定義し、それに基づいて設計
  - ただしこの２つの作業は、行ったり来たりしながら行われる;
    要求を定義する → 設計する → より詳細な要求を定義する → 詳細を設計する → …

cf. SE-J, 5.1.5




# Module

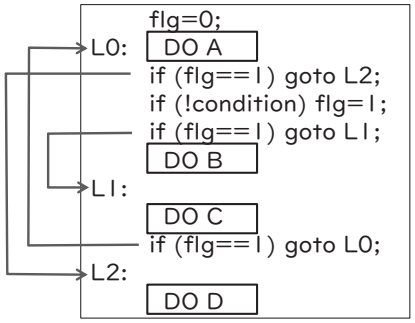



# モジュール

cf. SE-J, 5.2

# Structured programming

- Once upon a time, when computers were very expensive, ...
  - Programs with many "goto" statements.
    - One of the techniques to calculate fast with a small memory
    - Bad readability. Hard to improve.

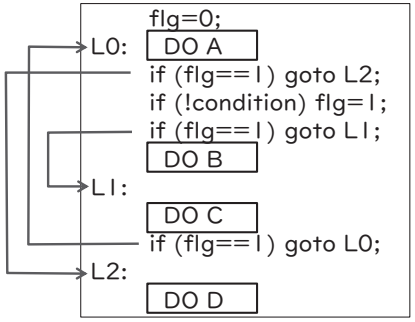






# 構造化プログラミング

- またコンピュータが非常に高価だった時...
  - プログラムは多くの"goto"文を持っていた
    - gotoを使うのは、少ないメモリで高速に処理するため
    - しかし読みにくく、修正・改造が難しい

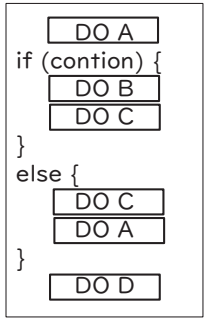






# Structured programming (cont.)

- In the late 1960s, the structured programming was introduced.
  - Less goto statements
  - Control structures
    - Sequence
    - Selection
    - Iteration

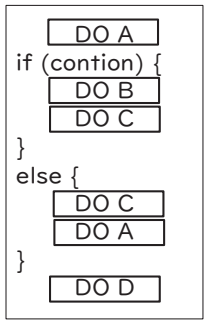



- In programming, separating control units and making their relations simple.




# 構造化プログラミング (Cont.)

- 1960年代後半、構造化プログラミングが提案された
  - goto文の多用をやめる
  - 基本3構造で整理する
    - 順次
    - 選択
    - 繰り返し




- プログラミングレベルにおいて制御の観点から要素単位に分けその間の関係性を単純にする

# Modularization

- Module
  - A composing element of software. A building block of software.
  - In other words, component; unit.
- Modularization
  - to define modules necessary to construct software and their relationships.
  - a key issue of software design.
  - has impacts on qualities.
  - affects the ease of division of work and the structure of development team, because each module becomes a unit of development works.



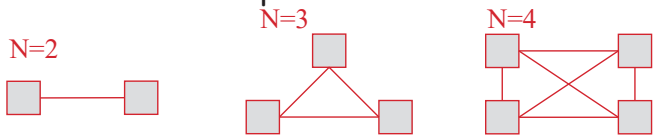

# モジュール化

- モジュール
  - ソフトウェアの構成要素
  - 別の言葉で言えば、コンポーネント、ユニット
- モジュール化
  - ソフトウェアを構成するために必要なモジュールとモジュール間の関係を定義
  - ソフトウェア設計の鍵
  - 品質に影響
  - それぞれのモジュールが開発単位になるので、開発分担の容易さや開発チームの構成に影響




# Dilemma of modularization

- If the number of modules are large,
  - each module can be small. → Easy to understand each module.
  - the number of relationships increases. → Difficult to understand impacts on many modules that one module has.
- If the number of modules are small,
  - each module can be large.
  - the number of relationships decreases.

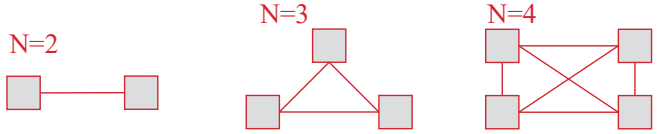






# モジュール化のジレンマ

- モジュールの数が多いと
  - それぞれのモジュールは小さくできる → それぞれのモジュールの理解は容易
  - 関連の数は増える → ひとつのモジュールが持つ他のモジュールへの影響の理解は困難
- モジュールの数が少ないと
  - それぞれのモジュールは大きくなりうる
  - 関係の数は減る



# Module cohesion and coupling

- Good modularization means that the software is constructed so that the change is localized and the influence of change is suppressed when software changes.
  - That means modules are independent.
- Module cohesion and module coupling are conceptual metrics of module independence.
  - conceptual = qualitative, not quantitative



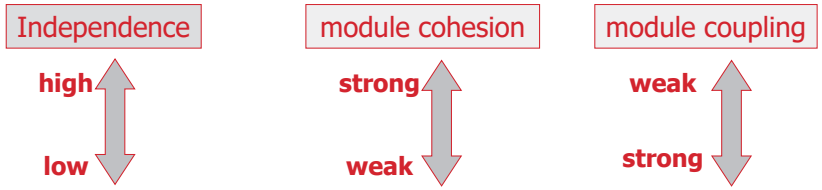

# モジュール強度・モジュール結合度

- 良いモジュール化とは、変更が局所化され、変更の影響が抑えられていること
  - モジュールが独立していることを意味する
- モジュール強度・モジュール結合度は、モジュールの独立性をはかる概念的な尺度
  - 概念的 = 定性的、定量的ではない




# Module cohesion and coupling

- Module cohesion: Strength of ties within a module.
  - The role that the module plays in a system is:
    clear → cohesion is strong.
    vague → cohesion is weak.
- Module coupling: Ties between modules.
  Relationships between modules.
  - Only necessary and sufficient relations exist.
    → coupling is weak.
    Unnecessary relations exist.
    → coupling is strong.

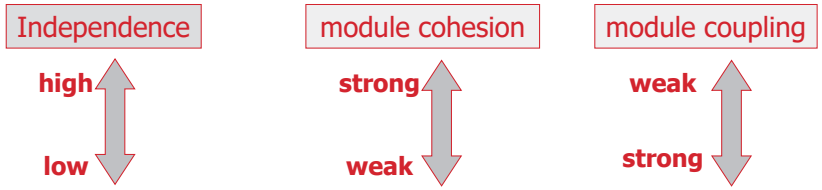






# モジュール強度・モジュール結合度

- モジュール強度: モジュール内の結びつきの強さ
  - システムがモジュールで果たす役割が
    明確 → 強度が強い
    あいまい → 強度が弱い
- モジュール結合度: モジュール間の結びつきの強さ、モジュール間の関係性
  - 必要かつ十分な関係しかない
    → 結合度が弱い.
    不必要な関係が存在する
    → 結合度が強い



# Information hiding

- Information hiding is to hide the details of the internal structure from the outside, to allow the outside to use only the operations published, and to increase the independence.
- Important concept used to weaken module coupling.



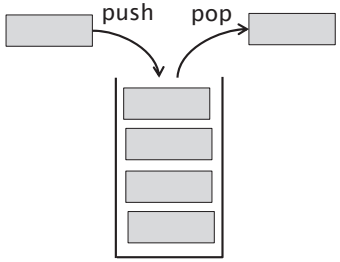

# 情報隠蔽

- 内部構造の詳細を外部から隠蔽し、外部からは公開された操作しかできないようにし、独立性を高めること
- モジュール結合度を弱めるために使われる




# Example: stack

- One of the famous data structure
- can store multiple pieces of data. The last stored data is retrieved first.
- Last In First Out (LIFO)

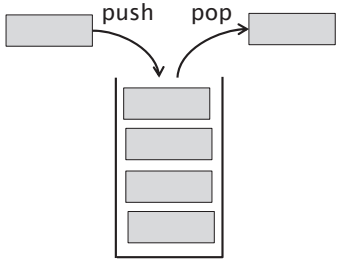



Published operations are "push" and "pop."
Users can only call these operations, and cannot access the data directly.
The detail of the data implementation is hidden.




# 例: スタック

- 有名なデータ構造のひとつ
- 複数のデータを格納
  最後に入れられたものが最初に取り出される
- Last In First Out (LIFO)




公開操作は "push" と "pop."
ユーザはこれらの公開された操作を呼ぶことしかできず、内部のデータに直接アクセスすることはできないデータ実装の詳細は隠されている

# Encapsulation

- a means of software development to separate functions and data of the system from other parts and to define the specification of those functions and data (such as the published operation).
  - to hide the inside which should not be seen from the outside.
- one of the mechanisms to ensure information hiding.




# カプセル化

- システムの機能やデータを他の部分から分離し、その部分の使用(公開操作等)を定義するためのソフトウェア開発上の手段
  - 見えてはならない内部を外部から見えなくする手段
- 情報隠蔽を実現する一手段




# Encapsulation (cont.)

- Example of encapsulation in Java
  - "class"

```
public class IntegerStack {
   private int[] data; // concrete data
   private int stackptr;

   public void push (int x) {
      data[stackptr] = x;
      stackptr++;
   }
   public int pop() {
      stackptr--;
      return data[stackptr];
   }
}
```




# カプセル化 (cont.)

- Javaにおけるカプセル化の例
  - "class"

```
public class IntegerStack {
   private int[] data; // concrete data
   private int stackptr;

   public void push (int x) {
      data[stackptr] = x;
      stackptr++;
   }
   public int pop() {
      stackptr--;
      return data[stackptr];
   }
}
```

## Data abstraction

- a process that focuses on the operation that can be done on data, and discards details such as how to express it.
- abstract data type is a data type defined as a set of operations.
- evolved to the object-oriented programming




## データ抽象

- データに対して何ができるのかという操作に注目し、その表現方法などの詳細を捨象するプロセスをデータ抽象という
- 抽象データ型とは、操作の集合として定義されたデータ型
- オブジェクト指向プログラミングに発展




## Viewpoints of software design




## 設計の視点

# Basic viewpoints of design

- How can we find and design modules to realize good modularization?
  - There is no unique answer. However, there are basic viewpoints of design and we choose and/or combine these viewpoints.

| Viewpoints | Typical methodologies | Typical diagrams & models |
|---|---|---|
| Function | Structured method<br>Process oriented approach | Data flow diagram<br>Structured chart |
| Information, data | Data oriented approach | Entity relationship diagram<br>Relational model |
| State | State transition design | State machine diagram<br>Sequence diagram |




# 設計の基本的な視点

- どのようにして良いモジュール化を実現?
  - 唯一の答えはない。しかし、設計の基本的な視点は存在し、それらから選んだり、組み合わせたりして、設計を行う

| 視点 | 典型的な手法 | 典型的な図やモデル |
|---|---|---|
| 機能 | 構造化手法<br>プロセス中心アプローチ | データフロー図<br>ストラクチャチャート |
| 情報, データ | データ中心アプローチ | 実体関連図<br>リレーショナルモデル |
| 状態 | 状態遷移設計 | ステートマシン図<br>シーケンス図 |




# Overview of structured method

- Method to structure software
- Focus on functions of software
- Generic name of various methods focusing software functions.
  - Structured design: Stevens(1974), Yourdon(1979)
  - Structured analysis: DeMacro(1979)
- Called also "process oriented approach"




# 構造化手法の概要

- ソフトウェアを構造化する手法
- ソフトウェアの機能にフォーカス
- ソフトウェアの機能にフォーカスする手法の総称
  - 構造化設計: Stevens(1974), Yourdon(1979)
  - 構造化分析: DeMacro(1979)
- "プロセス中心アプローチ"とも

cf. SE-J, 5.3

# Functional decomposition

- Modules in structured method are to realize functions and called as processes.
- Each function is a transformation of input to output.
  - Ex. Division is a data transformation from two inputs, divisor and dividend, to one output, quotient.
- In structured method, a system is considered as one function and is decomposed into necessary sub functions.
  → functional decomposition



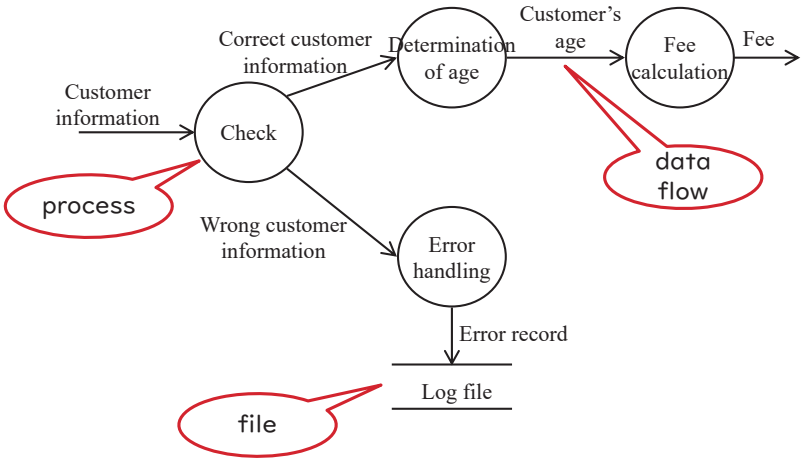

# 機能分割

- 構造化手法において機能を実現するモジュールはプロセスと呼ばれる
- それぞれの機能は、入力を出力に変換
  - 例. 除算は、2つの入力、つまり割られる数、割る数から1つの出力、つまり商への変換と捉えられる
    → 機能分割




# Data flow diagram

- Description of functional decomposition

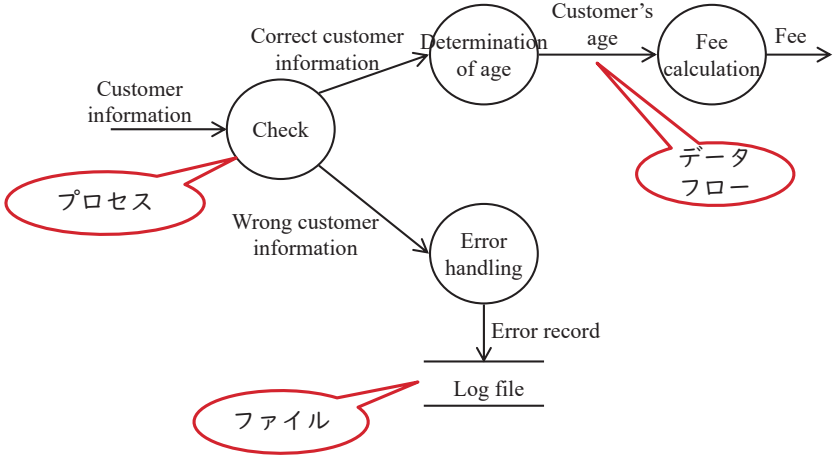




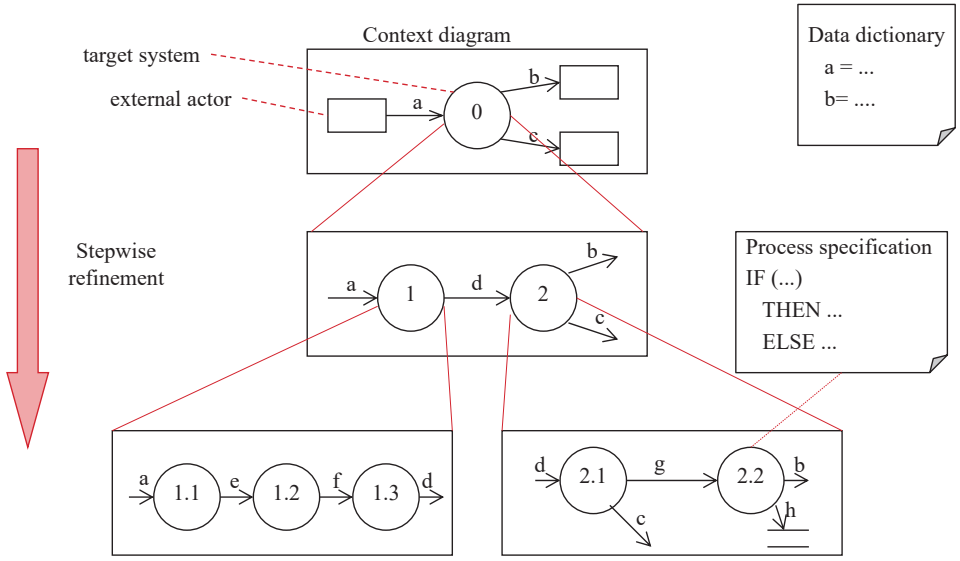


# データフロー図

- 機能分割の表現



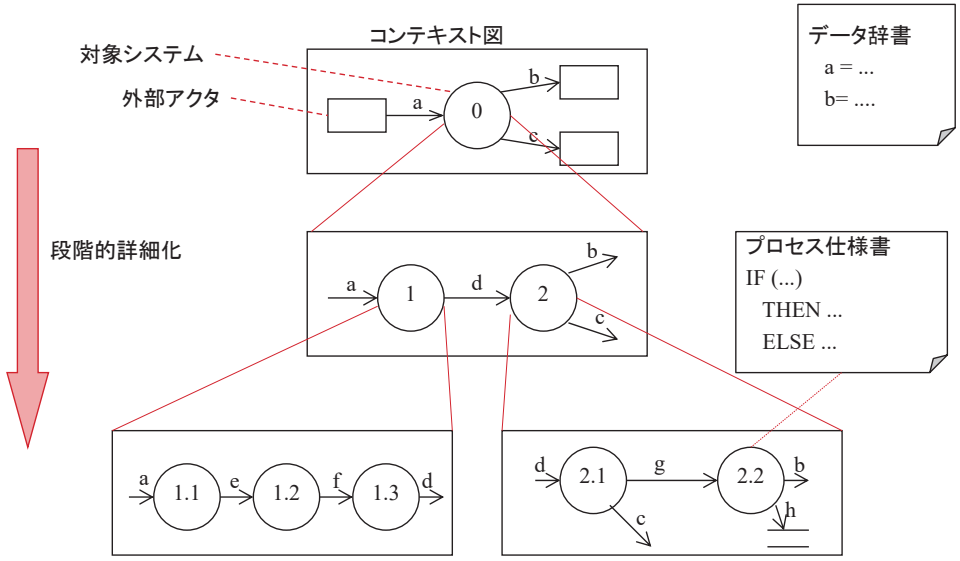
# Stepwise refinement







# 段階的詳細化

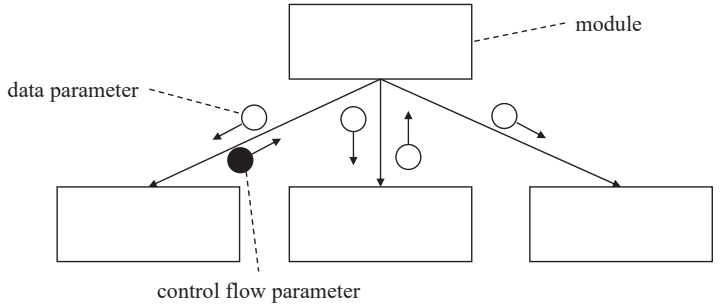






# Structured chart

- shows program structure
- In structured method, each process becomes a module, and a data flow becomes a relationship between modules.
  - ex. module = function, relationship = function call

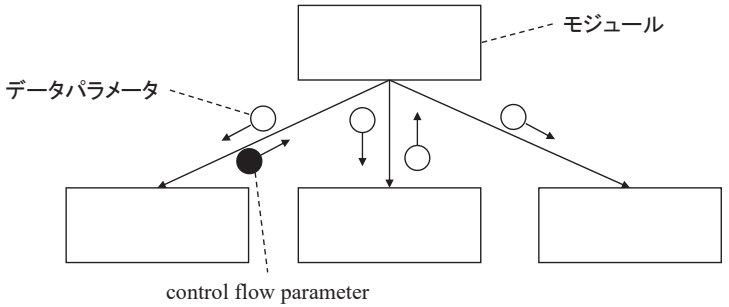






# ストラクチャチャート

- プログラム構造を示す
- 構造化手法では、それぞれのプロセスがプログラムモジュールになり、データフローはモジュール間の関係になる
  - 例. モジュール = 関数, 関係 = 関数呼び出し



# Data oriented approach

- The structure of information and data is clarified before the design of functionality.
- The structure of information and data
  → database design

For your preparation of the next class, please survey "entity relationship diagram" and "relational model". Next week, the overview of data-oriented approach will be explained.




# データ中心アプローチ

- 機能の設計をする前に、情報やデータの構造の明確化を行うアプローチ
- 情報やデータの構造
  → データベース設計

「実体関連図」と「リレーショナルモデル」について予習しておきましょう。
このアプローチの簡単な概要は来週説明します。

cf. SE-J, 5.4




# Design focused on state

- Many embedded systems are reactive systems.
  - reactive behavior means behavior that returns a response to an event.
- In a reactive system, each response is determined by combination of an event and the internal state.
- Design focused on state is necessary to design such reactive systems.




# 状態に注目した設計

- 多くの組込みシステムはリアクティブシステム
  - リアクティブなふるまいとは、イベントに対して反応を返すふるまい
- リアクティブシステムでは、それぞれの反応はイベントと内部状態で決まる
- このようなリアクティブシステムを設計するには、状態に注目した設計が必要

cf. SE-J, 5.5

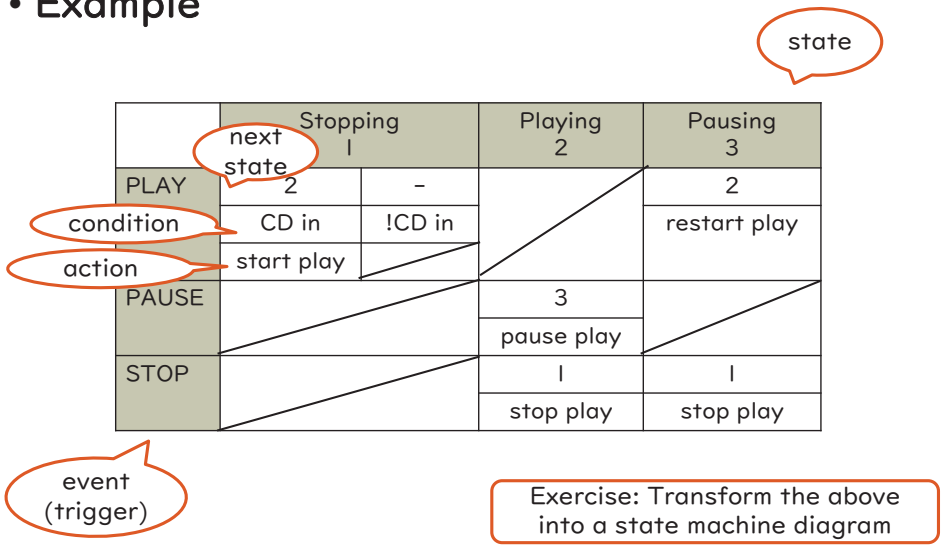
# Descriptions of state transitions

- State machine diagram
  - describes possible states and transitions between them graphically
  - easy to follow state transitions
- State transition table
  - describes states in a table.
    - row: state name
    - column: event name
    - cell: name of a next state on the top, activities on the transitions on the bottom
  - easy to find lacking states and/or transitions




# State transition table

- Example




Exercise: Transform the above into a state machine diagram



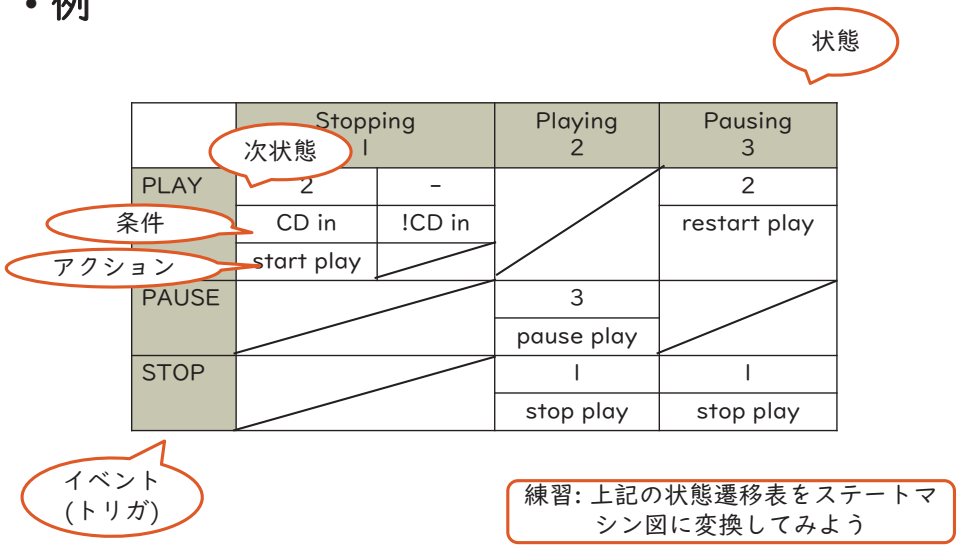

# 状態遷移の記述

- ステートマシン図
  - 起こり得る状態とその間の遷移をグラフィカルに表現
  - 状態遷移の把握が簡単
- 状態遷移表
  - 状態を表で表現
    - 行: 状態名
    - 列: イベント名
    - セル: 上段に次の状態の名前、下段に遷移上のアクティビティ
  - 状態や遷移の抜け・もれが見つけやすい




# 状態遷移表

- 例




練習: 上記の状態遷移表をステートマシン図に変換してみよう

# For your review

- List diagrams that are standardized by UML.
- Explain the relationship between requirements definition and designing.
- What are the 3 main categories of control structures of the structured programming?
- What are two conceptual metrics of module independence?
- List three basic viewpoints of designing good module structure.




# For your review

- Transform the state machine diagram below into a state transition table.