Classnotes 5:

1. Design and Information Flow

   - A *data flow diagram* (DFD) is a graphical technique that is used to depict information flow, i.e., a representation of information as a continuous flow that undergoes a series of transform (processes) as the information evolves from input to output.
   - DFDs may be partitioned into levels that represent increasing information flow and functional detail. At each level, additional processes is partitioned to reveal more detail.
     - A level 0 DFD, also called a *fundamental system model*, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively.
     - A level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 are sub-functions of the overall system.
   - Data flow-oriented design, i.e., "structured design", defines a number of different mappings that transform information flow into program structure.
     - Extended from techniques that stress on modularity, top-down design, and structured programming, this approach explicitly integrates information flow into the design process.
     - Such an approach is particularly useful when information is processed sequentially and no formal hierarchical data structure exists, such as microprocessor control applications, complex numerical analysis procedures, process control, and many other engineering and scientific software applications.

2. Two Information Flow Types

   - A transform flow is a segment of a DFD that consists of an *incoming flow*, a *transform center*, and an *outgoing flow*. The first one transforms external data into an internal form; the second transforms the incoming data into outgoing data; the last leads the transformed data "out" of this segment. The overall flow of data occurs in a sequential manner and follows one, or only a few, "straight-line" path.
   - Information may also be characterized by a single data item, called a *transaction*, that triggers other data flow along one of many paths. A *transaction flow* is presented when a DFD takes this form.

3. Design Process Considerations

   - The transition from information flow to structure is accomplished as part of a five-step process: (1) The type of information flow is established; (2) flow boundaries are indicated; (3) the DFD is mapped into program structure; (4) the control hierarchy is defined; (5) the resultant structure is refined using design measures and heuristics.
   - A Process Abstract: A design begins with an evaluation of the level 2 or level 3 data flow diagram. The information flow category (i.e., transform or transaction flow) is established, and flow boundaries that delineate the transform or transaction center are defined. Based on the location of boundaries, transforms (the DFD "bubbles") are mapped into program structure as modules.

```
Define level-0 data flow diagram;
Refine data flow diagram:
    If the type of flow is ‘‘Transform’’, do transform analysis:
        Identify incoming/outgoing branches;
        Map to transform structure.
    Else, the type of flow is ‘‘Transaction’’, do transaction analysis:
        Identify transaction center and data aquisition path;
        Map to transaction structure.
    Factor the structure;
    Refine the structure using design heuristics;
    Develop interface description and global data structure;
    Review;
Continue until reaching to the detail design.
```

4. An Example

   - the production description
     Our research indicates that the market for home security systems is growing at a rate of 40 percent per year. We would like to enter this market by building a microprocessor-based home security system that would protect against and/or recognize a variety of undersirable "situations" such as illegal entry, fire, flooding, and others. The product, tentatively called SafeHome*, will use appropriate sensors to detect each situation, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.*

Classnotes 6:

1. Rapid prototyping is primarily a requirements discovery technique, used to help determine the application functionality, data structure, and control characteristics of a system.

   - A software rapid prototype is a dynamic, interactive, visual model of the user's requirements as an implemented design.
     - User requirements are explored through experimental development, demonstration, refinement, and iteration.
     - The prototype provides a basis for dialog between developers and users that is far more effective than either text or static models.

2. The characteristics of a useful rapid prototype:

   - It is built quickly and demonstrated early.
   - It provides mechanism for users to try out proposed parts of a system, and then give direction for additional features and refinement.
   - It is easy to modify.
   - It is initially intentionally incomplete.

3. The risky suboptimal approaches

   - *Pretty Screens* does a little prototyping of the appearance of a few screens for the users.
     - Screens are shown and modified until pleasing; then the real screens are programmed to match.
     - User approval of such a prototype does not constitute any kind of meaningful requirements definition.
     - Evaluating the prettiness of a screen will not reveal whether the information shown on the screen is easy to obtain, accurate, sufficient, or intelligible.
     - Only dynamic prototype allows the user to experiment with these factors interactively, while viewing familiar data being manipulated.
   - *Only-a-model prototypes* use model only to help identify requirements during the analysis phase of a development project.
     - The prototype is built using an advanced development environment suitable for a quick build and rapid modifications.
     - After requirements are finalized, programming starts over from scratch using the real implementation language on the target machine.
     - This approach is often forced on developers who lack industrial-strength prototyping tools or do not have access to the target hardware.
       * Environments that do not produce software with good performance characteristics or allow the substitution of lower-level language programs for poorly performing prototype code cannot be used effectively for evolutionary prototyping.
       * Some factors that get in the way of acquisition of good tools are procurement constraints, organizational politics, and hardware preferences.
     - Only-a-model prototyping may not save much money in development costs, but it still provides the same benefit of reducing enhancement costs during the maintenance phase because the right requirements will be implemented.

- *Over-engineered prototypes*
  - A prototype is shown to users shortly before final delivery only to make sure that nothing has been overlooked.
  - It is not good to resist user-requested changes simply because they are inconsistent with a user-approved paper specification.
    * Users frequently approve paper specifications out of frustration with the lengthy specification process and lack of comprehensibility of the resulting product. Specification sign-off usually comes down to whether or not users trust their developers (experience teaches them they should not).
    * Often, developers of over-engineered prototypes offer users the opportunity to incorporate major change requests into a phase two project after delivery the current version of the software. This is a big waste of time and money because it usually means that major portions of the old requirements and design documents, as well as the prototype, must be rewritten. The smart rapid prototype avoids specifying features when not very certain of necessary or correct implementation.
    * What prototypes do best is serve to *invalidate* pre-specified requirements by uncovering all of the user-developer misunderstandings.
- *hacking*
  - Sometimes rapid prototyping is used as an excuse for not doing any form of requirements or design specification, or at least an amount well below what is required for a high-quality, maintainable system.
    * In hacking, developers work hard, fast, and iteratively on the code alone, until they get it right.
    * The users may or may not be asked to review the various evolving versions.
    * It's rapid because they work really hard and because they escape the requirements to document because they're supposedly using a state-of-the-art rapid prototyping approach.
  - This approach used to be called hacking as it does not improve the quality of the results.
    * In the case of hacking, modifying the programs many times during prototype iterations will eventually destroy any readability or understandability that was designed into the initial version.
    * If design specifications are not produced first, the programs will probably not be easy to read and understand.

Classnotes 7:

1. SOFTWARE DESIGN

   Deriving a solution which satisfies software requirements

2. Stages of design

   - Problem understanding
     - Look at the problem from different angles to discover the design requirements
   - Identify one or more solutions
     - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources
   - Describe solution abstractions
     - Use graphical, formal or other descriptive notations to describe the components of the design
   - Repeat process for each identified abstraction until the design is expressed in primitive terms

3. The design process

   - Any design may be modelled as a directed graph made up of entities with attributes which participate in relationships
   - The system should be described at several different levels of abstraction

- Design takes place in overlapping stages. It is artificial to separate it into distinct phases but some separation is usually necessary

4. Phases in the design process
5. Design strategies
    - Functional design
        - The system is designed from a functional viewpoint. The system state is centralised and shared between the functions operating on that state
    - Object-oriented design
        - The system is viewed as a collection of interacting objects. The system state is de centralised and each object manages its own state. Objects may be instances of an object class and communicate by exchanging methods
6. Functional view of a compiler
7. Object-oriented view of a compiler
8. Design quality
    - Design quality is an elusive concept. Quality depends on specific organisational priorities
    - A 'good' design may be the most efficient, the cheapest, the most maintainable, the most reliable, etc.
    - The attributes discussed here are concerned with the maintainability of the design
    - Quality characteristics are equally applicable to function-oriented and object-oriented designs
9. Cohesion
    - A measure of how well a component 'fits together'
    - A component should implement a single logical entity or function
    - Cohesion is a desirable design component attribute as when a change has to be made, it is localised in a single cohesive component
    - Various levels of cohesion have been identified
10. Cohesion levels
    - Coincidental cohesion (weak)
        - Parts of a component are simply bundled together
    - Logical association (weak)
        - Components which perform similar functions are grouped
    - Temporal cohesion (weak)
        - Components which are activated at the same time are grouped
    - Procedural cohesion (weak)
        - The elements in a component make up a single control sequence
    - Communicational cohesion (medium)
        - All the elements of a component operate on the same input or produce the same output
    - Sequential cohesion (medium)
        - The output for one part of a component is the input to another part
    - Functional cohesion (strong)
        - Each part of a component is necessary for the execution of a single function
    - Object cohesion (strong)

– Each operation provides functionality which allows object attributes to be modified or inspected

11. Coupling

- A measure of the strength of the inter-connections between system components
- Loose coupling means component changes are unlikely to affect other components
- Shared variables or control information exchange lead to tight coupling
- Loose coupling can be achieved by state decentralisation (as in objects) and component communication via parameters or message passing

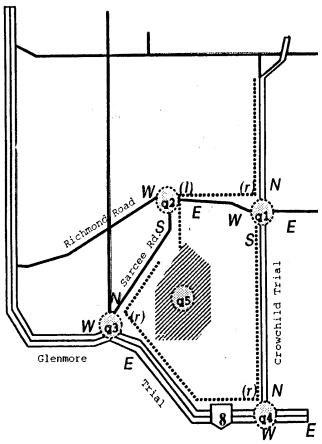
Classnotes 8:

(i) general system model:

$$\begin{cases} function\ elelment \rightarrow information\ transformers\ in\ the\ system. \\ data \rightarrow inputs\ and\ outputs\ of\ functions. \\ control \rightarrow the\ mechanism\ that\ actives\ functions\ in\ the\ desired\ sequence. \end{cases}$$

(ii) *finite states machine* is appropriate for the modeling of real-time systems where a particular input causes a thread of actions to be initiated.

- `Webster's Dictionary:`
  An *automaton* is a machine or control mechanism designed to *follow automatically a predetermined sequence of operations* or respond to encoded instructions.
- `Components:`
  - *Input Channel* $\mathbf{X}$ reads in $\{x_1, ..., x_m\}$.
  - *Output Channel* $\mathbf{Y}$ sends out $\{y_1, ..., y_n\}$.
  - *Output Operators* $\mathbf{\Phi}$ generate $y_s = \mathbf{\Phi}(q_i, x_r)$.
  - *Internal States* $\mathbf{Q}$ indicate $\{q_1, ..., q_k\}$.
  - *State Operators* $\mathbf{\Psi}$ change q_i to $q_j = \mathbf{\Psi}(q_i, x_r)$.
- `Definition:`
  An *automaton* is quintuple $< \mathbf{Q}, \mathbf{X}, \mathbf{Y}, \mathbf{\Psi}, \mathbf{\Phi} >$, where $\mathbf{Q}$ has finite elements. Under $qx \rightarrow \mathbf{\Psi}(q, x)\mathbf{\Phi}(q, x)$, which has recurrence relations as $q(t+1) = \mathbf{\Psi}[q(t), x(t)]$ and $y(t) = \mathbf{\Phi}[q(t), x(t)]$, with an initial condition $q(1) = q_0$, an automaton operates at discrete sampling time $t = 1, 2, ...$ to transforms every finite sequence of input symbols $x = x(1)x(2)x(3)...x(r)$ into a same length sequence of output symbols $y = y(1)y(2)y(3)...y(r)$.
- `Operation (Road Map):`

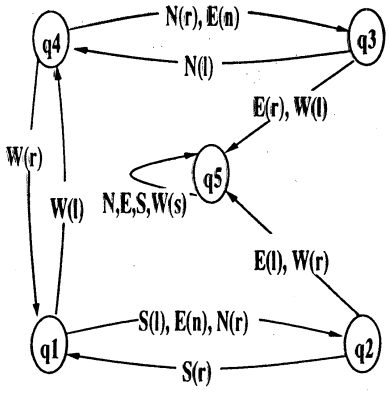


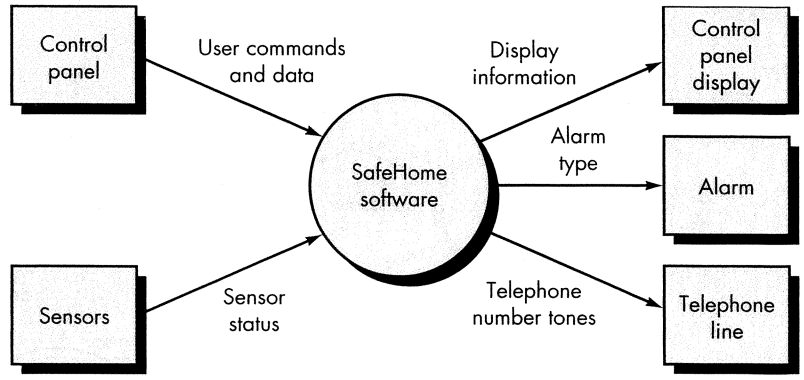
  - Input alphabet $\mathbf{X}$={N, E, S, W}
  - Output alphabet $\mathbf{Y}$={r, l, n, s}
  - Internal Alphabet $\mathbf{Q}$={q_1, q_2, q_3, q_4, q_5}

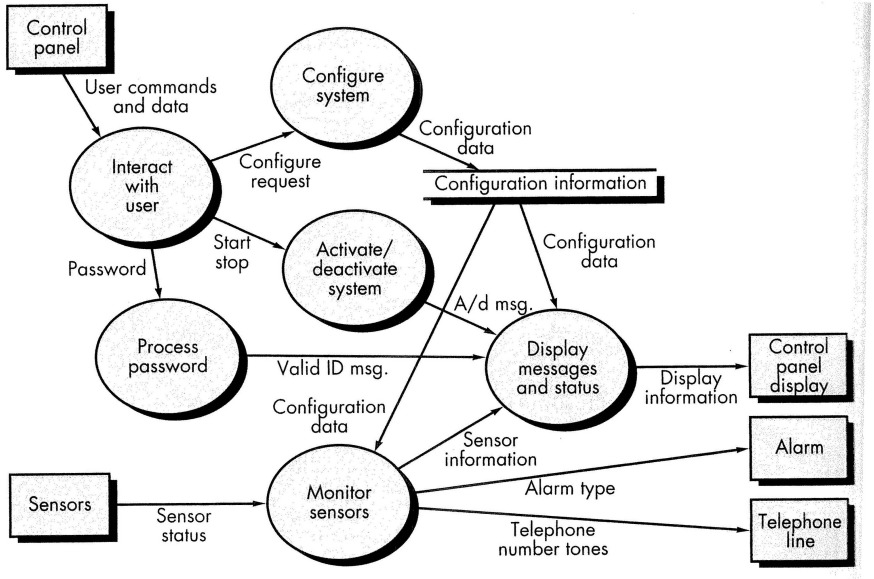
– Operators Φ and Ψ

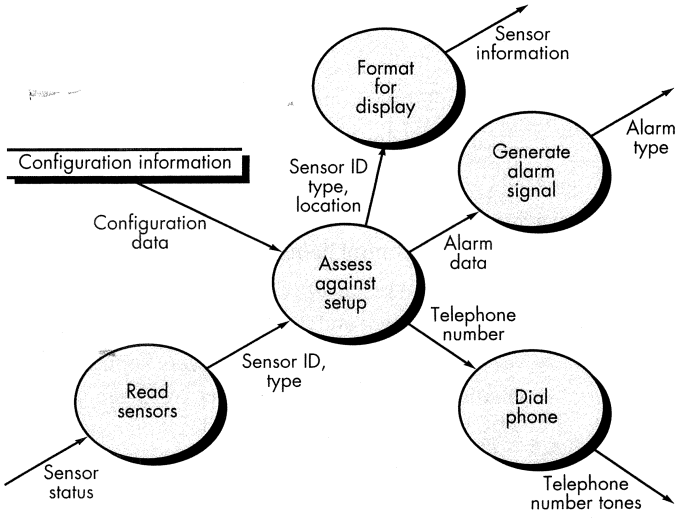
`state` q_i	`input` x_i	`output` $y_i = \Phi(q_i, x_i)$	`new state` $q_{i+1} = \Psi(q_i, x_i)$
q_1	N	r	q_2
q_2	E	l	q_5
q_1	N'	n	q_4
q_4	N	r	q_3
q_3	E	r	q_5
$\vdots$			$\vdots$
q_5	N,E,S,W	s	q_5

- `A diagram`




Control panel
User commands and data
Display information
Control panel display
SafeHome software
Alarm type
Alarm
Sensors
Sensor status
Telephone number tones
Telephone line


Control panel
User commands and data
Configure system
Configuration data
Interact with user
Configure request
Configuration information
Start stop
Activate/ deactivate system
Configuration data
Password
A/d msg.
Process password
Valid ID msg.
Display messages and status
Control panel display
Display information
Configuration data
Sensor information
Alarm
Sensors
Monitor sensors
Alarm type
Sensor status
Telephone number tones
Telephone line


Format for display
Sensor information
Generate alarm signal
Alarm type
Configuration information
Sensor ID type, location
Configuration data
Assess against setup
Alarm data
Telephone number
Sensor ID, type
Read sensors
Dial phone
Sensor status
Telephone number tones