SE 2A04 Fall 2000

04. Design Techniques

Instructor: W. M. Farmer

Revised: 30 October 2000

Design and Complexity

- Many naive design techniques fail when applied to complex systems
- Useful software systems are inherently complex
  - There are many necessary details
  - The margin of error is often extremely small
- Engineers and mathematicians have developed effective techniques for managing complexity which can be applied to software development

Techniques for Managing Complexity

- Modularization (horizontal decomposition)
- Abstraction (vertical decomposition)
- Definition
- Special purpose languages
- Information hiding
- Axiomatic method
- Abstract machines
- Prototyping

Software Design Models

1. Traditional (data and procedures)
2. Modules
3. Object oriented
4. Finite state machines
5. Streams
6. Communicating processes

1. Traditional Model

- The model has two kinds of entities:
  1. Objects and information are represented as **data**
  2. **Procedures** are descriptions of processes that create and modify data
- Major issue: How are the data and procedures organized?
  - Horizontal grouping: packages, modules, objects
  - Vertical grouping: local data and procedures
- Important variant:
  - Procedures themselves are treated as data
  - Supported by Lisp, ML, Scheme, etc.

Modules

- A **module** is a cohesive collection of data and procedures that provide a set of **services** to other modules
- A module consists of an **interface** for exchanging information with other modules and an **implementation** of the interface that is hidden from other modules
- The interface is a **language** for requesting the services
  - The primitive components are usually procedures called **interface functions** or **access functions**
- The implementation provides the services
  - The interface functions are implemented together and may share data structures
  - The implementation may utilize the services offered by other modules

Hallmarks of Good Module

- The module is as independent from other modules as possible
- The set of interface functions is small and orthogonal
- The interface language is highly expressive
- Implementation details are hidden from other modules (**information hiding** or **encapsulation**)
- The data structures of the implementation are accessible only via the interface functions

2. Module Model

- A software product is decomposed into a small collection of modules
- Design decisions that are likely to change are hidden from other modules (**design for change**)
- Benefits:
  - Modules can be developed independently
  - Modules can be reused and interchanged
  - The implementation of a module can be freely changed
  - Modules can hold a state

Objects

- Object have two kinds of components:
  - Data (**fields**)
  - Procedures (**methods**)
- Objects have **state**
  - The values of an object's fields determine its state
- Objects have **behavior**
  - The implementation of an object's methods determine its behavior
- An object can be viewed as:
  - A **record** with procedures for accessing the fields
  - A **module**

Object Oriented Principles

- Objects communicate via **message passing**
- The behavior of **generic operators** is distributed across different objects
- A collection of objects with similar structure is specified by a **class**
- Objects are created by **instantiating** a class
- A **subclass** C' of a class C **inherits** the structure and behavior of C

3. Object Oriented Model

- A software product is decomposed into a collection of objects
- Since an object is a special kind of module, the object oriented model has all the benefits of the module model
- The object oriented model is more structured than the module model
- An object oriented software design can be implemented using a non-object-oriented programming language

Finite State Machines

A **finite state machine** M consists of the following components:

1. A fixed finite set S of **states** including an **initial state**
2. A fixed (possibly infinite) set I of **inputs**
3. A fixed (possibly infinite) set O of **outputs**
4. A **next state** relation ns $\subseteq S \times I \times S$
5. An **output** relation out $\subseteq S \times I \times O$

The finite state machine is very useful as both a practical and theoretic tool

Example: “aba” Recognizer

$S = \{1, 2, 3\}$ and the initial state is 1

1 means “ ” has been recognized

2 means “a” has been recognized

3 means “ab” has been recognized

$I = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

$O = \{\mathtt{y}, \mathtt{n}\}$

ns	a	b	c
1	2	1	1
2	2	3	1
3	2	1	1

out	a	b	c
1	n	n	n
2	n	n	n
3	y	n	n

What is a State?

- A state of a machine is an abstract entity that can only be defined indirectly
- A **description of a state** of a machine is a description of all the information needed to predict the machine's future response to input from the external environment
- Physical machines have an infinite number of states, but they can usually be viewed as if they had a finite number of states
  - Aspects of a state which are irrelevant to the behavior of the machine (e.g., temperature and location) can be ignored
  - **Transition states** between **stable states** can also be ignored
- Digital computers are design to behave as if they were finite state machines

Some Finite State Machine Definitions

- A finite state machine is **deterministic** if ns and out are functions, i.e., ns : $S \times I \to S$ and out : $S \times I \to O$
- Two deterministic finite state machines are **equivalent** if, given the same sequence of inputs, both machines produce the same sequence of outputs
  - That is, they are equivalent if their “black box” behavior is the same
  - Equivalent finite state machines may not have the same number of states
- Two states are **equivalent** if they are indistinguishable from outside the machine
- A finite state machine is **minimal** if no two of its states are equivalent

4. Finite State Machine Model

- A software product is designed as a finite state machine
  - Care is required in the selection of the set of states
- The finite state machine model helps the designer make sure that no important cases have been overlooked
  - Especially if tables are used to describe ns and out
- The complexity of a finite state machine can be reduced by decomposing the machine into a **network of finite state machines**
  - The output of one machine is the input of another
  - An output can be split into several inputs
  - Several outputs can be merged into a single input
  - The same input can be sent to more than one machine

5. Stream Model

- A **stream** is a (possibly infinite) sequence of data
- Streams can serve as a common interface for combining procedures
- Since a stream can represent the time history of a system, the Stream Model provides a way of modeling systems that have state without using mutable data
- Efficient implementation of streams requires **delayed** or **lazy evaluation**
  - The production of a stream is interleaved with the consumption of the stream

6. Communicating Processes Model

- A software product is a collection of independent processes that communicate over one-way channels
- Examples:
  - Cooperating programs running as processes on an operating system
  - An application distributed over a computer network
- Processes normally run concurrently, which leads to several issues not found in sequential programming
- An important special case is the **Client-Server Model**
  - Server processes provide services to a network
  - Client processes utilize services offered to a network