**Design process**
• Requirements
• Design
• Development
• Testing

**Characteristics of Good Design**
• Ease of understanding
• Ease of implementation
• Ease of testing
• Ease of modification
• Satisfies requirements

**Modularity and Abstraction**
• Characteristic of all design methods
• Components have clearly defined inputs and outputs, and clearly stated purpose
• Easy to examine each component separately
• Organized so system can be investigated one part at a time

***client***
programmers who will use our software
develops software that will call our functions
***coworkers***
programmers who will debug and upgrade our software
develops, tests, and modifies our software.

**Hierarchical or Layered Design**
– Components at one level refine those in the level above
– More abstract top levels hide detail of lower level

**Modules**
In C we equate module with program file
*Example*: TxFifo in project SCIA
1) Collection of functions
2) Data structures and data types
Header file is the function prototypes (what it does)
Comments written for the user or customer
Defines the interface, data types
How functions are called

What the functions do
What values are returned?
Code file is the implementation (how it works)
Comments written for ourselves or our coworkers
Helps to assist in debugging, and extending
How the functions were tested?
What assumptions were made?
Variables should be private (make them local, or use static)

**Information Hiding**
• Components hide internal details and processing from one another
• Each hides a specific design decision
• Allows isolation complex parts of problem, where "decisions" need to be made
• Possible to use different design methods within different components
• When a hidden design decision changes, only that component must be changed
• The rest of the design remains intact

**How do we hide information?**
• Do not put it in header file
• Private functions
• Private variables

**Modularity**
Maximize the number of modules
Minimize the coupling between modules

**Coupling**
• Highly coupled = great deal of dependence between components
• Loosely coupled = some dependence but interconnections are weak
• Uncoupled = no interconnections at all

**How components are coupled**
• References from one component to another, such as invocation
• Bandwidth of data passed from one component to another
• Amount of control one component has over another

**Types of Coupling (from bad to good)**
• *Content:* one component directly modifies data or control flow of another

• *Common:* Organizing data into a common store
• *Control:* Control flags passed as parameters between components
• *Stamp:* Data structures passed
• *Data:* Only primitive data passed

**Maximize Cohesion**
• Cohesion: the degree of interrelatedness of internal parts.
• All elements of the component are directed towards and essential for the same task

**Degrees of Cohesion (from bad to good)**
• ***Coincidental****:* component's parts unrelated
• ***Logical****:* logically related functions and/or data placed in same component
• ***Temporal****:* performs several functions in sequence, related only by timing (not ordering)
• ***Procedural****:* functions grouped together to ensure mandatory ordering
• ***Communicational****:* functions operate on or produce same data set
• ***Sequential****:* output from one function is input to next
• ***Functional****:* every processing element essential to single function, and all such essential elements contained in one component

**Fans, observe the data flow graph**
• ***Fan-in****:* number of components controlling a particular component
• ***Fan-out****:* number of components controlled by a given component
• Minimize components with high fan-out
–Probably doing too much, performing more than one function
• High fan-in should be limited to utility components
**• Goal – High fan-in; low fan-out**

**Exceptions**
• "Defensive design": actively anticipate situations that may lead to problems
• Make explicit what the system is **not** supposed to do
– Not easy to define
• Should do what it is required to do "but no more"
– Not easy to define or test
• *Exception*: situation counter to what system should do.

**Typical exceptions:**
– Failure to provide service or data
– Providing wrong service or data
– Corrupting data

**Exception Handling**
• ***Retry:*** restore system to previous state and try again using a different strategy
• ***Correct:*** restore to previous state, correct some aspect, and try again (same strategy)
• ***Report:*** restore to previous state, report problem, and do not provide service

Rewrite TxFifo.c TxFifo.h for the following cases
0) change mechanisms without changing policies
use indices instead of pointers
use hardware instead of software
increase the fifo size when full
1) 8-bit to 16-bit to 32-bit
2) clone the code for additional fifos

**Dynamic efficiency**
**Bandwidth**
- **calculations performed per second**
- **data transferred per second**

**Latency or response time (real time means bounded latency)**
- **time from a request to the time action satisfies request**
- **time new input ready to time data is read**
- **time output device is idle to time new data is written**

**Static efficiency**
**RAM (variables) and ROM (constants, program)**
**9S12DP512 14 kibibytes RAM, 512 kibibytes of EEPROM**
**9S12C32, 2 KiB RAM, 32 KiB EEPROM**
**9S12C128, 4 KiB RAM, 128 KiB EEPROM**
**MSP430F2013, 128 bytes RAM, 2 KiB EEPROM**

**attitude**
- embarrassed to deliver poorly written software

- modules that are easy to change
- expect our code will be modified
- reward good behavior
- punish bad behavior
- test it now, fix it now
- plan for testing