Design

○ In software engineering, refers to the process of describing **how** requirements will be met.
○ Without actually doing it! The latter is **implementation**.

Reading: SEPA Chapter 8.
Phases of design

**Overall design:** describes the pieces that can be completed independently.
  Top-level: identifies the **main modules**
  Refined: identifies **submodules** that comprise a module.

**Detailed design:** describes the inner details of each module.
Design Goals

Divide the project into reasonably independent chunks

Describe points of interaction between chunks.

Enable parallel development of chunks by different teams.

Be specific on what programmers must do to create a chunk.

Assign chunks to different developers/teams.

Balance labor between developers/teams.

Overall goals:

Maximize cohesion: a chunk should do one thing.

Minimize coupling: chunks should interact minimally.

Balance labor: work assignments should minimize total development time.
Real reason for design: to *decouple* teams from one another.

- Design serves as a *conduit* for information between teams working on different chunks.
- So teams don't have to *communicate (much).*
- Unless the design is *incomplete* or *poor.*
Another reason design is important: outsourcing
- American company creates design.
- Design is sent to outsourcing company in the third world where programmer labor is cheaper (or perhaps -- an external consultant in the US).
- Outsourcing company returns programming product based upon design (and perhaps sells it in other countries!)
- Unless design is complete, result will not be desirable.
A really good design ... 

precisely satisfies requirements as to what the product should do.

leaves no doubt as to the expectations for the final product.

enables parallel development of the chunks comprising the design.

Employs (and creates) reusable software components.
Coupling is bad
Wednesday, October 07, 2009
11:50 AM

Coupling leads to
increased communication during implementation.
limits on parallel development.
increased constraints on evolution of the product.
in extreme cases, inability to adapt to changing needs.

Cohesion leads to
Less to remember for programmers.
Leveraging existing knowledge more effectively.

Work imbalances lead to
Longer development time.
Longer time to market.
A **module** is a unit of software that it is practical to:

- Develop separately.
- Test separately.

Modules can have **submodules** that solve part of a module's problem.

A **module decomposition** is a description of how to break a complex project into modules.

Typically, this is expressed as a tree of modules:
Why modules are important

- Enable breaking up a complex task into subtasks.
- Allow testing of individual pieces separately from one another.

A module is a **unit of delegation:**
- Modules allow programmer independence in completing subtasks.
- Submodules allow teams to further delegate tasks to individual programmers.
Why is coupling such a problem?
- The cost of software is proportional to the amount of communication needed to produce it.
- Brooks' law: "Adding manpower to a late project makes it later."
- Minimizing coupling minimizes necessary communications

A coupling horror story: Apple-II
- vendor: published whole source to operating system.
- programmers: utilized completely weird entry points into the software (e.g., making subroutine calls into the middle of a subroutine).
- result: effective code freeze of OS.
- long-term result: death of Apple-II

lesson learned: MacOS:
- no source code.
- "jump table" indicates usable system calls.
- use these or else "you'll be sorry"

MS-DOS BIOS: same story, different players!
What is a module?
- a unit of code that can be updated independently of other modules.
- specific unit varies with programming language
- In linux C and C++, the module unit is a source code file.
- In C# and java, the module boundary is a class.
In detail: modules in C++

Unit of independent work is a file. Files can contain multiple classes and/or functions. Files are extremely difficult to co-edit at the same time. Projects are made up of individual files and libraries.
How a large C++ project is written:

Break up into **individual files** (modules).
Write a **header file** for each file.
**Include** this header in other files that need it.
Write a large compilation command that links all files.
Create a **Makefile** that automates the process.
Use **Automake** to write portable Makefiles.
A definition creates a tangible thing in memory
  Function
  Variable
  int foo(int i) { return i*2; }
  Cannot appear in header files.
A declaration describes the nature of a thing (that is created separately).
  extern int foo(int);
  Can appear in header files.
Ideally, foo.h declares what is defined in foo.cc

What is
  inline int foo(int i) { return i*2; }
  This is a declaration, because the code does not create anything in itself. It's a macro.
  Compiler translates foo(j) to j*2 wherever it appears, without calling a subroutine.
#define FOO 1
  It's a declaration, because no memory is created.
First step in design: **module decomposition**

**express coding problem** as a tree or graph

**partition the graph** so that the minimum number of edges are required between partitions.

Make each partition a **module** (whose submodules are the nodes in the partition)

**Assign each module** to a developer or team.

Use some form of **workload estimation** to check fairness of assignment.
The design process

- **modeling**: description of the components of a design.
- **refinement**: the process of adding detail to a depiction.
- **partitioning**: the process of grouping cohesive functionality into modules.
- **assignment**: assigning modules to people; forming programming teams.
- **workload estimation**: determining how difficult each part will be to write.

Many design methodologies

- Jackson Systems Design (JSD)
- Data Structured Systems Design (DSSD) *
- Dataflow Design (DFD) *
- Object-oriented Design (OOD) *

Every design methodology has

- Diagramming techniques
- Methods for transforming diagrams into decompositions.
- Strengths and weaknesses.

Which one to use?

- efficiency of modeling/depiction.
- efficiency of refinement
- efficiency of compartmentalization and
modularization

- match between design methodology and problem domain.
DSSD: structure of a program follows that of the data structures it manipulates.
DFD: structure of a program mimics data transformation and flow patterns.
OOD: structure of program mimics class relationships between objects.
In JSD,
- Structure of data is parallel to structure of program.
- Module decomposition is a matter of combining connected components of the Jackson tree, so that coupling is minimal.

Jackson's classic example: accounts payable
Control modules

In a module decomposition, often there are modules that do nothing more than to act as "glue". We call these "control modules".

Purpose of control modules

delegate communications to one team in a star configuration.
points of the star aren't loaded down with communications.
Control modules are join points in a Jackson diagram:

- If we decompose each node into a module, A and B are "control modules."
- Could also decompose so that A, B are in some module.
- Or even so C0, C are in some module.

Design Page 21
Every design methodology assumes that some commensurability between tasks we're partitioning a set of roughly equal things.

What can go wrong? what if one process, not decomposed, is much more difficult than the others?
Data Structured Systems Design (DSSD) (Warner-Orr diagrams)

- more "modernized" JSD
- new concept: the **data dictionary**
  - "nouns": data or systems
  - "verbs": methods
- Jackson's principle still applies
  - represent data relationships as a hierarchical list.
  - utilize data structure/control structure duality.
  - Partition based upon hierarchy.
First principle: data structure is parallel to control structure.

Second principle: data flow parallels control flow.

DFD: dataflow design
- diagram interactions using data flow graph.
  - transaction flow: commands are moving
  - transform flow: data is moving
- partition dataflow graph into sections with minimal coupling = maximal cohesion. Separate regions of transform and transaction flow.
- use synthetic control modules to simulate data flow in a control flow environment.
DFD example:

A control program for an automated manufacturing facility has several parts whose combined task is to manufacture a part from a schematic. First, a raw schematic and a database of the properties of materials are used to create a better schematic that can be manufactured. Based upon the contents of this new schematic and a database of the capabilities of milling machines, one of three kinds of milling machines is selected to make the part. The schematic is then compiled into a control program for this kind of milling machine. Finally, the control program is used to operate the machine and make the part.
Transform flow

Thursday, November 10, 2005
2:08 PM
Transaction flow

Thursday, November 10, 2005
2:08 PM

A \rightleftharpoons B
A \rightleftharpoons C
A \rightleftharpoons \text{"transactions"}
If I have

A → B → C

A → C → F → H

D → G → control

A

B C D E F G H
Transaction centers

Monday, September 17, 2012
5:28 PM
End of lecture 9/17/2012
Monday, September 17, 2012
5:32 PM
Transform flow: an item is moving

e.g. \( x = \text{foo}(y) \) means

\[
\begin{align*}
\text{unbilled account} & \quad \longrightarrow \quad \text{foo} \\
\text{x} & \quad \longrightarrow \quad \text{foo} \\
\text{billed account} & \quad \longrightarrow \quad \text{foo}
\end{align*}
\]

Transaction flow

"go" \( \longrightarrow \) foo() \( \longrightarrow \) "done"

"accounts receivable" \( \longrightarrow \) "accounts payable"
Dataflow design steps

- model transform(data) and transaction(control) flow within a system
- identify regions of data and/or transaction flow.
  - Transaction flow:
    - Identify **transaction center** = group of operations that control whole process.
    - Treat this as top-level module.
    - Inputs and outputs of transaction center form submodules.
  - Transform flow:
    - Identify **transform center** = group of operations that control transformation process.
    - Treat these as top-level module.
    - Inputs and outputs form submodules.
Ambiguity:

Module B calls A

B

\[ \text{depiction of } B \]

C

\[ \text{depiction of } C \]

Depends upon complexity

A \rightarrow B \rightarrow C

\[ \text{classical recomputation} \]

\[ \text{synthetic} \]
Amoiguites cont'd

foo = A(cat)  bar = B(foo)  dog = C(bar)

cat -> A -> B -> C -> dog

C_1 = \frac{3}{2}

\text{defn:}
\begin{align*}
  \text{foo} &= A(\text{cat}) \\
  \text{bar} &= B(\text{foo}) \\
  \text{dog} &= C(\text{bar}) \\
  \text{return} \text{ dog;}
\end{align*}
Combine CFD and DFD. Depict couplings via "Booch Diagrams". Blur distinction between data flow and control flow. Partition for maximum cohesion, minimal coupling.

A rebarancy
interface

C \rightarrow B.

from somewhere in A, rel. E
Inheritance is just another information domain.

There is a natural relationship between objects in the real world and their representations in a program.

Objects in the real world obey conditions:
- More conditions => subclass.
- More conditions => less instances.

Objects in a program have attributes:
- More attributes => subclass.
- More attributes => less instances.

Basic principle of inheritance: identifying subclass relationships maximizes code reuse.
Unified modeling language (UML) provides one way to diagram multiple information domains. Data flow, control flow, containment, inheritance.
\[ A \leftarrow B \quad B \text{ refers to } A \]

\[ A \leftarrow B \]

B refers to a list of A's

\[ B \]

\[ B \quad \text{is an A (inheritance)} \]

\[ A \]

An A contains a list of B's.
Two notations: \[ O \leq \begin{cases} & 1 \quad \begin{cases} & + \quad \# \end{cases} \\
& 0 \quad \begin{cases} & + \quad \# \end{cases} \end{cases} \]

Inheritance:

- Card
- Acet holder

Container:

- Business
defect
- Entity
- Aut +
- Entity plus
- Acet

\[ \hspace{1cm} \]
General principles of module decomposition

Segregate the kinds of information flow
transaction flow
data flow
inheritance (condition flow)

Seek one or more "centers" from which to decompose modules
transaction center
transform center
root class

Build a basic module decomposition from the "centers"
Refine that decomposition by creating submodules for the centers.
After you have a module decomposition, you must assign it to programmers and teams.

Basic issue: how "fair" is the assignment? For that matter, what is the difficulty of programming a specific module? A long story, and our next topic.