CSE 403
Lecture 12

Software Decomposition

Design principles

- Understand the basic principles underlying software design
  - Modularization
  - Coupling
  - Cohesion
- Understand that the driving force behind design is managing complexity
- Provides a basis for studying information hiding, layering, patterns, etc.

What is design?

- The activity that leads from requirements to implementation
- A description that represents a key aspect of this activity
- If the requirements are the "what" the design (with an associated implementation) is the "how"

Design space

- There are many designs that satisfy a given set of requirements
- There are also many designs that may at first appear to satisfy the requirements, but don't on further study
- Collectively, these form a design space
- A designer walks this space evaluating designs

Design: managing complexity

- Of course, this design space isn't just sitting out there for you to search like a library catalog or the WWW
- You must also generate the designs
- A key aspect of design generation is understanding that the goal is to achieve the requirements in the face of the limitations of the human mind and the need for teams to develop products

Dijkstra

- From "Programming considered as a human activity"
  - "The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and rule). … I assume the programmer's genius matches the difficulty of his problem and assume that he has a arrived at a suitable subdivision of the task."
- This final assumption is troubling! It's a big part of the job of software design!

Motivation for Modules

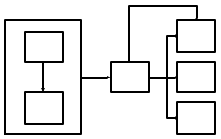
- Managing complexity
- Independent development and maintenance
- Reuse
  - Component reuse
  - Application reuse
    - Portability
    - Versioning

Decomposition

- Design is largely a process of finding decompositions that help humans manage the complexity
  - Understand that the design satisfies the requirements
  - Allow relatively independent progress of team members
  - Support later changes effectively
- Not all decompositions are equally good!

Decompositions

- A decomposition specifies a set of components (modules) and the interactions among those modules
  - It is often the case that the components are related to parts of an object model
- The degree of detail in the specification varies



Aside: Composition

- Decomposition
  - The classic view of design
  - Given a fixed set of requirements, what decomposition is best?
- Composition
  - An increasingly common view of design
  - Given a set of available components, what design that exploits them is best?
  - Are there slightly different requirements that are easier to achieve given those components?

In software, far less is understood about composition-based design than needs to be

Comparing designs

- Not all decompositions are equally good
- So, on what basis do we compare and contrast them?
- Indeed, on what basis do we even generate them?
- Parnas said: “Usually nothing is said about the criteria to be used in dividing the system into modules.”

Coupling and cohesion

- Given a decomposition of a system into modules, one can partially assess the design in terms of cohesion and coupling
- Loosely, *cohesion* assesses why the elements are grouped together in a module
- Loosely, *coupling* assesses the kind and quantity of interconnections among modules

Kinds of cohesion

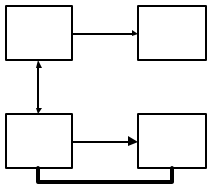
- Elements can be placed together to provide an abstract data type
  - if they are all executed about the same time (say, during initialization or termination)
  - because they will be assigned to a single person
  - because they start with the same letter
  - because...

"Good" vs. "bad" cohesion

- Best: functional, where the elements collectively provide a specific behavior or related behaviors
- Worst: coincidental, where the elements are collected for no reason at all
- Many other levels in between
- Cohesion is not measurable quantitatively

Coupling

- Coupling assesses the interactions between modules
- It is important to distinguish *kind* and *strength*
  - kind: A calls B, C inherits from D, etc.
    - And directionality
  - strength: the number of interactions



"Good" vs. "bad" coupling

- Modules that are loosely coupled (or uncoupled) are better than those that are tightly coupled
- Why? Because of the objective of modules to help with human limitations
  - The more tightly coupled are two modules, the harder it is to think about them separately, and thus the benefits become more limited

How to assess coupling?

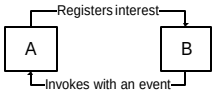
- Kinds of interconnections
- Strengths of interconnections
- There are lots of approaches to quantitatively measuring coupling
  - I'm not especially satisfied by any of them
  - They are not broadly used in industry
    - More used in reengineering than forward engineering
  - They are beyond the scope of this class

What kind (of relations)?

- There are many kinds of interconnections (relations) to consider
  - calls, invokes, accesses, ...
  - how about invokes in an OO language using dynamic dispatch?
  - others?
- Question: how many different relations are there among components of a software system?

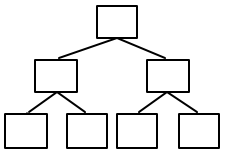
names vs. invokes

- Here is an example of a mix of relations
- Module A registers interest with an event that B can announce
- When B announces that event, a function in A is invoked
- A knows the name of B, but B doesn't know the name of A



Hierarchical designs

- Loose coupling is often associated with hierarchical designs
  - They also tend to arise from repeated refinements of a design
- Hierarchies are often more coupled than they appear
  - Because of other relations



Strength of interconnection

- Q: does a module that relies on an ADT module have stronger interconnection if it makes calls to all the exported methods (as opposed to only a couple of them)?
- Q: what if it calls one or two methods, but it calls them millions of times each?
- Q: what if it has more kinds of relations with that module?

Use a single module?

- Great cohesion!
- No coupling!
- Where's the failure?

Language support for modules

- Ada
- Basic
- C
- C++
- Fortran
- Java
- Pascal
- VB

How do languages support modularity

- Grouping
- Access levels
- Interfaces

Advantages of language support

- Facilities
  - Ease of expression of modules
- Regularity
  - Common mechanisms used for modularity
- Enforcement
  - Can count on mechanisms working (sometimes)
  - Error/violation detection

Faking modularity

- Manual packaging
- Naming conventions

- Modular programs can be written in languages that don't support modularity
- Non modular programs can be written in languages that do