# Design

CSE 403, Spring 2004

Software Engineering

http://www.cs.washington.edu/education/courses/403/04sp/



## References


» *Programming Considered as a Human Activity*, EW Dijkstra, Proceedings of the IFIP Congress 65

» *On the Criteria To Be Used in Decomposing Systems into Modules*, DL Parnas, Comm. of the ACM, V15, No 12, Dec 1972

- http://www.acm.org/classics/

» *The Hundred-Year Language*, Paul Graham

- http://www.paulgraham.com/hundred.html

» *Structure and Interpretation of Computer Programs (SIPC)*, Abelson & Sussman




## Design principles

- The driving force behind design is managing complexity
- "Programs must be written for people to read, and only incidentally for machines to execute."
  - » *SIPC*, Abelson & Sussman
- A basis for studying information hiding, layering, patterns, etc.
- The basic principles underlying software design
  - » Modularization
  - » Coupling
  - » Cohesion

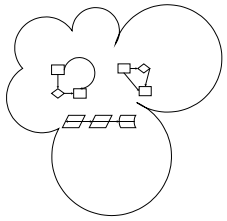


## What is design?

- The activity that leads from requirements to implementation
- If the requirements are the "what" then 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

- This design space isn't just sitting out there for you to search like a library catalog or the WWW
  - » You must generate the designs, using the appropriate language (and I don't mean "#@*&&!")
- A key aspect of design generation is understanding that the goal is to build a system that meets the requirements
  - » despite the limits of the human mind
  - » despite the limits of communication in large groups
  - » despite the limits of time and money



# Dijkstra: *Quality of Results*

- We should ask ourselves the questions:
  - » When an automatic computer produces results, why do we trust them, if we do so?
  - » What measures can we take to increase our confidence that the results produced are indeed the results intended?
- The programmer's situation is closely analogous to that of the pure mathematician, who develops a theory and proves results.
  - » One can never guarantee that a proof is correct; the best one can say is "I have not discovered any mistakes."
  - » … we do nothing but make the correctness of our conclusions plausible. So extremely plausible, that the analogy may serve as a great source of inspiration.



# Dijkstra: *Structure of Convincing Programs*

- The technique of mastering complexity has been known since ancient times: *divide et impera* (divide and rule).
  - » The analogy between proof construction and program construction is, again, striking.
  - » In both cases the available starting points are given; … the goal is given; … the complexity is tackled by division into parts.
- I assume the programmer's genius matches the difficulty of his problem and assume that he has arrived at a suitable subdivision of the task.

# Dijkstra: *Dissection*

- Proceed in the following stages
  - » make a complete specification of the parts
  - » show the problem is solved by the specified parts
  - » build the parts that satisfy the specifications, independent of one another and their context
- The technique relies on what I should like to call "The principle of non-interference."
  - » take into account the exterior specifications only
  - » not the particulars of their construction



# Boole: *Unity and Harmony*

- An Investigation of the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities
  - » Of the Conditions of a Perfect Method

I do not here speak of that perfection only which consists in power, but of that also which is founded in the conception of what is fit and beautiful. It is probable that a careful analysis of this question would conduct us to some such conclusion as the following, *viz*., that a perfect method should not only be an efficient one, as respects the accomplishment of the objects for which it is designed, but should in all its parts and processes manifest a certain unity and harmony.



# Graham: Language Core

- Any programming language can be divided into two parts: some set of fundamental operators that play the role of axioms, and the rest of the language
- I think it's important not just that the axioms be well chosen, but that there be few of them. Mathematicians have always felt this way about axioms-- the fewer, the better-- and I think they're onto something.

The Hundred-Year Language



# Motivation for Modules

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

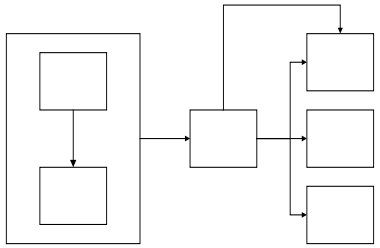
# Decomposition

- 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!



# Decomposition

- 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





# 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?



# Philosophy of Modular Programming

A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module… each module and its inputs and outputs are well-defined … the integrity of the module is tested independently … system errors and deficiencies can be traced to specific system modules.

*Designing Systems Programs*, Gauthier and Pont

## Comparing designs

- Not all decompositions are equally good
  - » on what basis can we compare and contrast them?
  - » on what basis can we even generate them?
- What are the criteria to be used in dividing the system into modules?
  - » each module is a step in the processing?
  - » each module is characterized by its knowledge of a design decision which it hides from all others?
    - Its interface or definition is chosen to reveal as little as possible about its inner workings.



## Parnas: *Decomposing Systems into Modules*

- Almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart (ie, control flow)
- Begin with a list of difficult design decisions or design decisions which are likely to change
  - » each module hides such a decision from the others
  - » since design decisions transcend time of execution, modules will not correspond to steps in the processing



## 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 are placed together to provide an abstract data type
- This is the appeal of object oriented design
  - » The internal state and the actual implementation of the class methods are together and hidden from external view
  - » The external world knows only the interface
  - » The internal world knows everything it needs to know to implement the desired behavior
    - and it knows nothing else

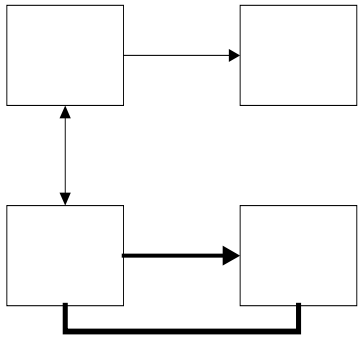
# "Good" vs. "bad" cohesion

- 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
  - » do these functions and ideas "belong together"?



# 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 work with them separately, and thus the benefits become more limited



# How to assess coupling?

- Types and strengths of interconnections
- There are lots of approaches to quantitatively measuring coupling
  - » No single number that decisively indicates good or bad
  - » But you can still get useful information about your code
    - JavaNCSS counts Non Commenting Source Statements (NCSS), packages, classes, functions and inner classes, and calculates Cyclomatic Complexity Number
    - JDepend traverses Java class and source file directories and generates design quality metrics for each Java package
    - both can be downloaded and can be run with Ant

# Choose your metrics ...

| SOURCE | METRIC | OBJECT-ORIENTED CONSTRUCT |
|---|---|---|
| Traditional | Cyclomatic complexity (CC) | Method |
| Traditional | Lines of Code (LOC) | Method |
| Traditional | Comment percentage (CP) | Method |
| NEW Object-Oriented | Weighted methods per class (WMC) | Class/Method |
| NEW Object-Oriented | Response for a class (RFC) | Class/Message |
| NEW Object-Oriented | Lack of cohesion of methods (LCOM) | Class/Cohesion |
| NEW Object-Oriented | Coupling between objects (CBO) | Coupling |
| NEW Object-Oriented | Depth of inheritance tree (DIT) | Inheritance |
| NEW Object-Oriented | Number of children (NOC) | Inheritance |

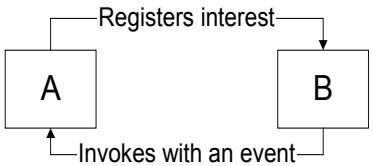
**Table 1: SATC Metrics for Object Oriented Systems**

NASA Software Assurance Technology Center (SATC) http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html



# Interconnections: names vs. invokes

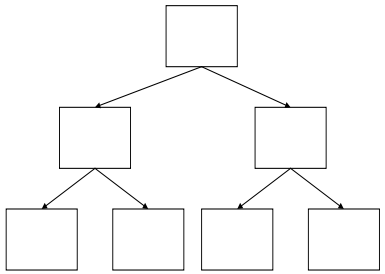
- Here is an example of a mix of relations
- Module A registers interest with an event that B can announce
- When B announces that event, a function in A is invoked
- A knows the name of B, but B doesn't know the name of A






# 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





# 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



## "Good" vs. "bad" waste

- I learned to program when computer power was scarce ... The thought of all this stupendously inefficient software burning up cycles doing the same thing over and over seems kind of gross to me. But I think my intuitions here are wrong. I'm like someone who grew up poor, and can't bear to spend money even for something important, like going to the doctor.
- There's good waste, and bad waste. I'm interested in good waste -- the kind where, by spending more, we can get simpler designs. How will we take advantage of the opportunities to waste cycles that we'll get from new, faster hardware?
- Most data structures exist because of speed.

Graham: The Hundred-Year Language



## More power supports better abstraction

- Another way to burn up cycles is to have many layers of software between the application and the hardware.
- This too is a trend we see happening already: many recent languages are compiled into byte code.
  - » Bill Woods once told me that, as a rule of thumb, each layer of interpretation costs a factor of 10 in speed. This extra cost buys you flexibility.

Graham: The Hundred-Year Language



## Elegance for better living

- The key to performance is elegance, not battalions of special cases.
  - » Jon Bentley and Doug McIlroy
- Premature optimization is the root of all evil (or at least most of it) in programming.
  - » Donald Knuth

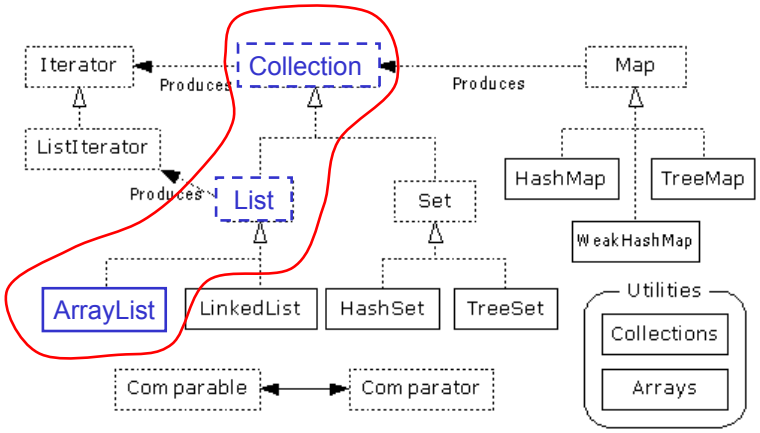
# Lisp: World Domination?

- Cobol, for all its sometime popularity, does not seem to have any intellectual descendants. It is an evolutionary dead-end -- a Neanderthal language. I predict a similar fate for Java.
- I don't predict the demise of object-oriented programming, by the way. Though I don't think it has much to offer good programmers, except in certain specialized domains, it is irresistible to large organizations. Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches.

Graham: The Hundred-Year Language



# Collections Framework Diagram




- Interfaces, Implementations, and Algorithms
- From Thinking in Java, page 462