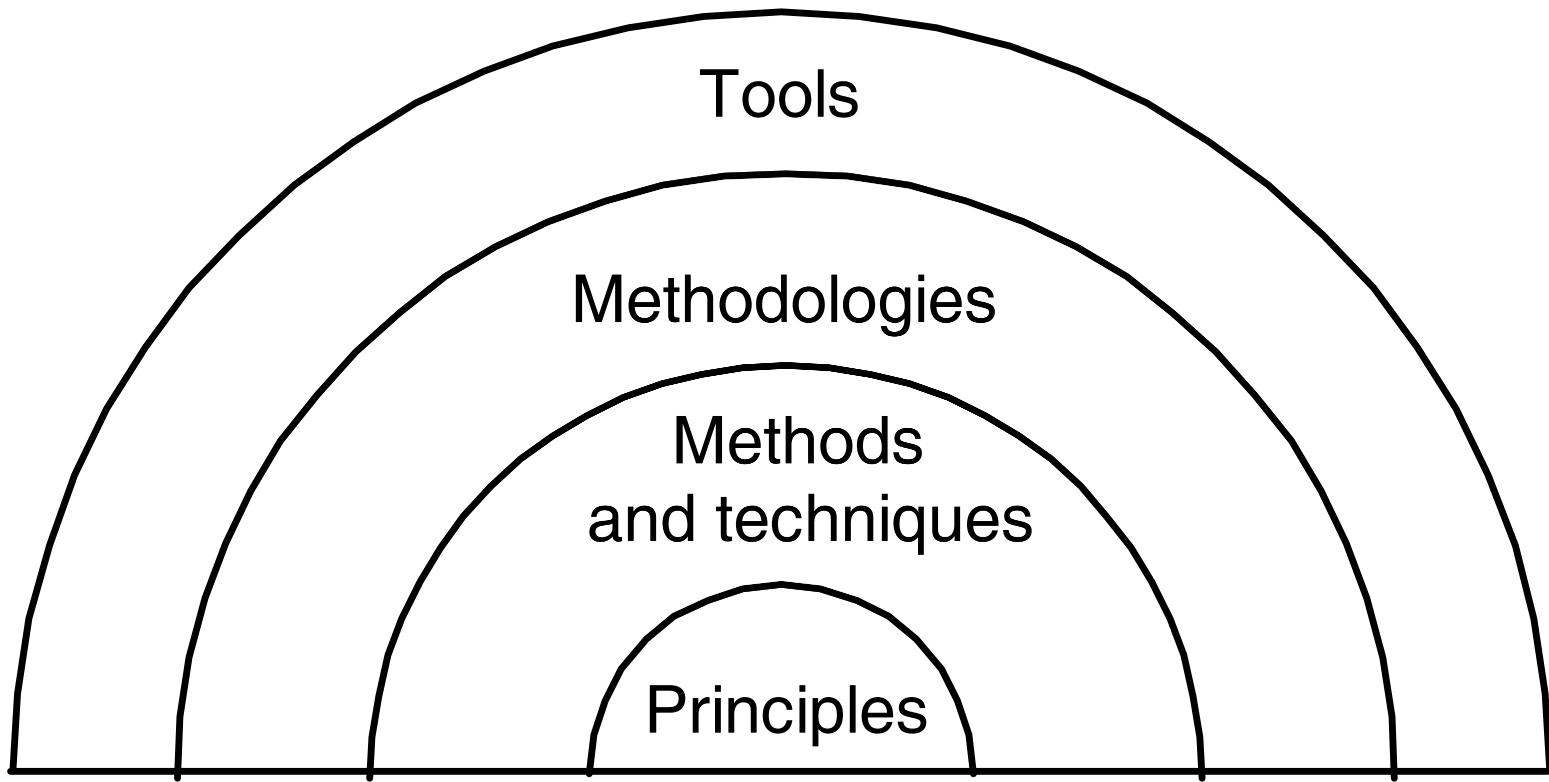
Outline

- Principles form the basis of methods, techniques, methodologies and tools
- Seven important principles that may be used in all phases of software development
- Modularity is the cornerstone principle supporting software design
- Case studies

Application of principles

- Principles apply to process and product
- Principles become practice through methods and techniques
  - often methods and techniques are packaged in a *methodology*
  - methodologies can be enforced by *tools*

A visual representation



Key principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Rigor and formality

- Software engineering is a creative design activity, BUT
- It must be practiced systematically
- Rigor is a necessary complement to creativity that increases our confidence in our developments
- Formality is rigor at the highest degree
  - software process driven and evaluated by mathematical laws

Examples: product

- **Mathematical (formal) analysis of program correctness**
- **Systematic (rigorous) test data derivation**

Example: process

- Rigorous documentation of development steps helps project management and assessment of timeliness

Separation of concerns

- To dominate complexity, separate the issues to concentrate on one at a time
- "Divide & conquer" (*divide et impera*)
- Supports parallelization of efforts and separation of responsibilities

Example: process

- Go through phases one after the other (as in waterfall)
  - Does separation of concerns by separating activities with respect to time

Example: product

- Keep product requirements separate
  - functionality
  - performance
  - user interface and usability

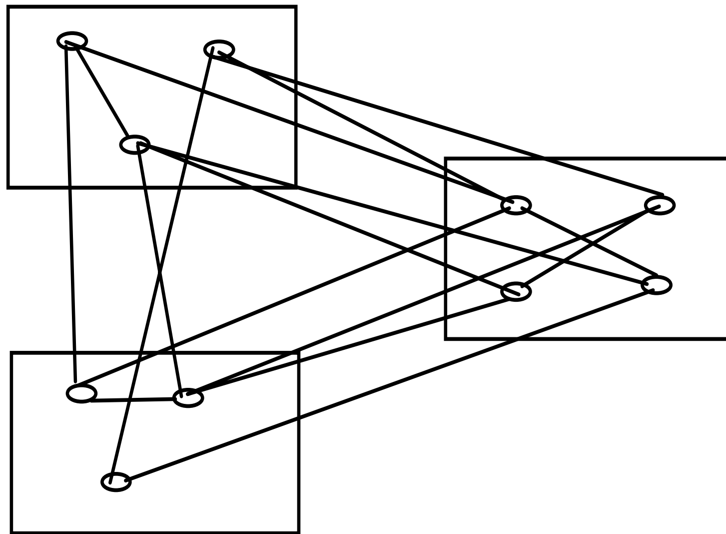
Modularity

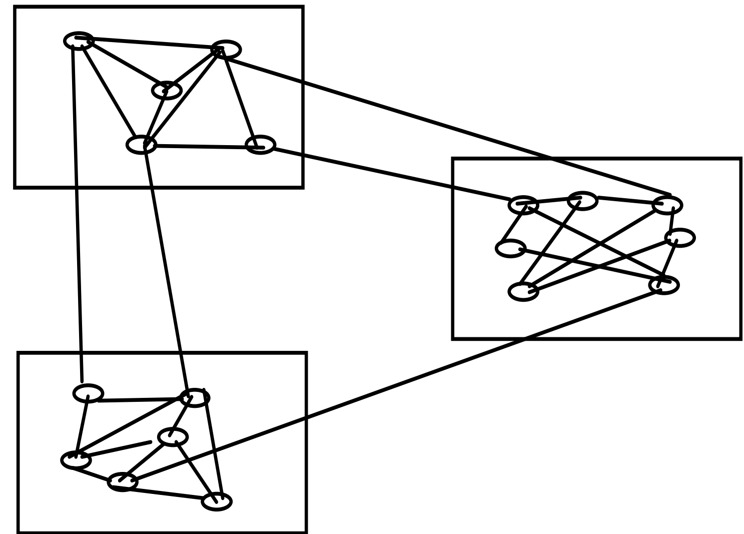
- A complex system may be divided into simpler pieces called *modules*
- A system that is composed of modules is called *modular*
- Supports application of separation of concerns
  - when dealing with a module we can ignore details of other modules

Cohesion and coupling

- Each module should be *highly cohesive*
  - module understandable as a meaningful unit
  - Components of a module are closely related to one another
- Modules should exhibit *low coupling*
  - modules have low interactions with others
  - understandable separately

A visual representation





high coupling

low coupling

Abstraction

- Identify the important aspects of a phenomenon and ignore its details
- Special case of separation of concerns
- The type of abstraction to apply depends on purpose
- Example : the user interface of a watch (its buttons) abstracts from the watch's internals for the purpose of setting time; other abstractions needed to support repair

Abstraction ignores details

- Example: equations describing complex circuit (e.g., amplifier) allows designer to reason about signal amplification
- Equations may approximate description, ignoring details that yield negligible effects (e.g., connectors assumed to be ideal)

Abstraction yields models

- For example, when requirements are analyzed we produce a model of the proposed application
- The model can be a formal or semiformal description
- It is then possible to reason about the system by reasoning about the model

An example

- Programming language semantics described through an abstract machine that ignores details of the real machines used for implementation
  - abstraction ignores details such as precision of number representation or addressing mechanisms

Abstraction in process

- When we do cost estimation we only take some key factors into account
- We apply similarity with previous systems, ignoring detail differences

Anticipation of change

- Ability to support software evolution requires anticipating potential future changes
- It is the basis for software evolvability
- Example: set up a configuration management environment for the project (as we will discuss)

Generality

- While solving a problem, try to discover if it is an instance of a more general problem whose solution can be reused in other cases
- Carefully balance generality against performance and cost
- Sometimes a general problem is easier to solve than a special case

Incrementality

- Process proceeds in a stepwise fashion (*increments*)
- Examples (process)
  - deliver subsets of a system early to get early feedback from expected users, then add new features incrementally
  - deal first with functionality, then turn to performance
  - deliver a first prototype and then incrementally add effort to turn prototype into product

Case study: compiler

- Compiler construction is an area where systematic (formal) design methods have been developed
  - e.g., BNF for formal description of language syntax

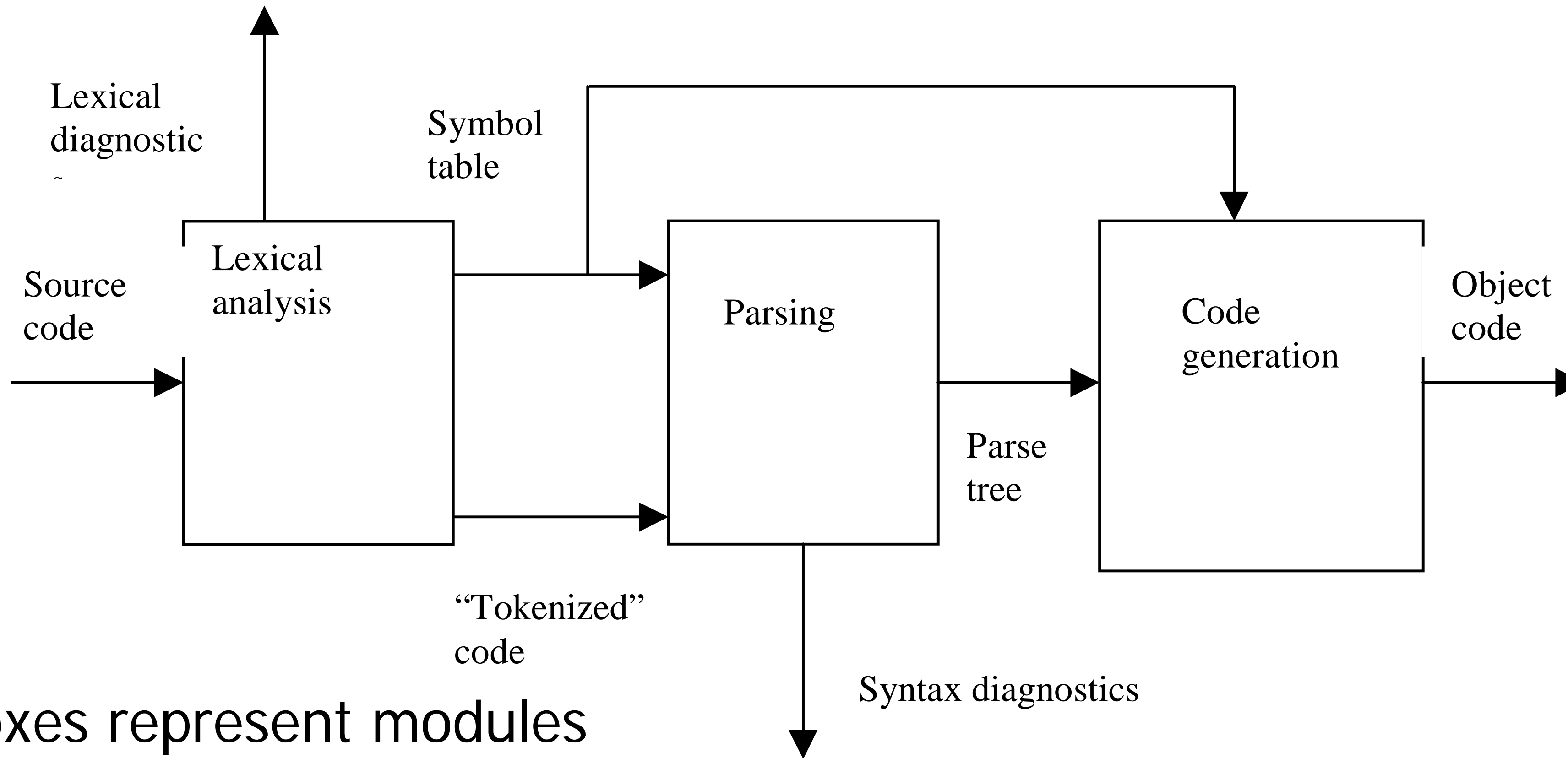
Separation of concerns example

- **When designing optimal register allocation algorithms (*runtime efficiency*) no need to worry about runtime diagnostic messages (*user friendliness*)**

Modularity

- Compilation process decomposed into phases
  - Lexical analysis
  - Syntax analysis (parsing)
  - Code generation
- Phases can be associated with modules

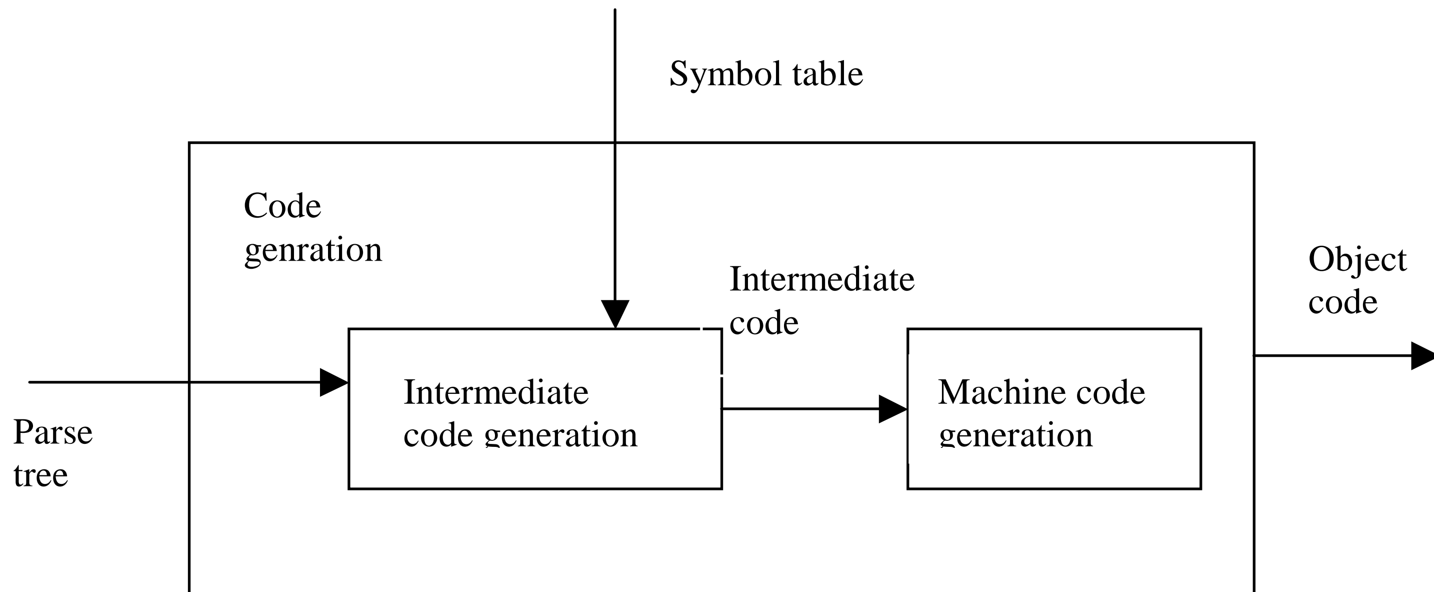
Representation of modular structure



boxes represent modules
directed lines represent interfaces

Module decomposition may be iterated

further modularization of code-generation module



Abstraction

- **Applied in many cases**
  - **abstract syntax to neglect syntactic details such as** begin…end **vs. {…} to bracket statement sequences**
  - **intermediate machine code (e.g., Java Bytecode) for code portability**

Anticipation of change

- Consider possible changes of
  - source language (due to standardization committees)
  - target processor
  - I/O devices

Generality

- Parameterize with respect to target machine (by defining intermediate code)
- Develop compiler generating tools (*compiler compilers*) instead of just one compiler

Incrementality

- Incremental development
  - deliver first a kernel version for a subset of the source language, then increasingly larger subsets
  - deliver compiler with little or no diagnostics/optimizations, then add diagnostics/optimizations

Case study (system engineering): elevator system

- In many cases, the "software engineering" phase starts after understanding and analyzing the "systems engineering" issues
- The elevator case study illustrates the point

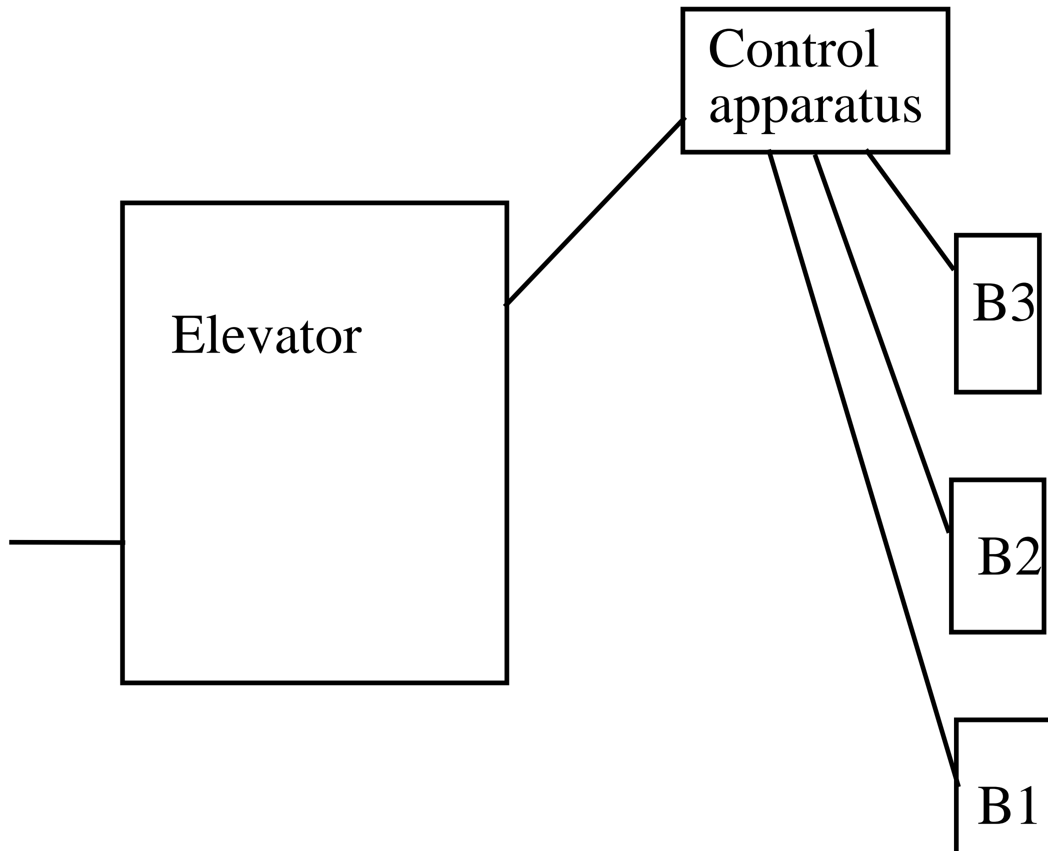
Rigor&formality (1)

- Quite relevant: it is a *safety critical system*
  - Define requirements
    - must be able to carry up to 400 Kg. (safety alarm and no operation if overloaded)
    - emergency brakes must be able to stop elevator within 1 m. and 2 sec. in case of cable failures
  - Later, verify their fulfillment

Separation of concerns

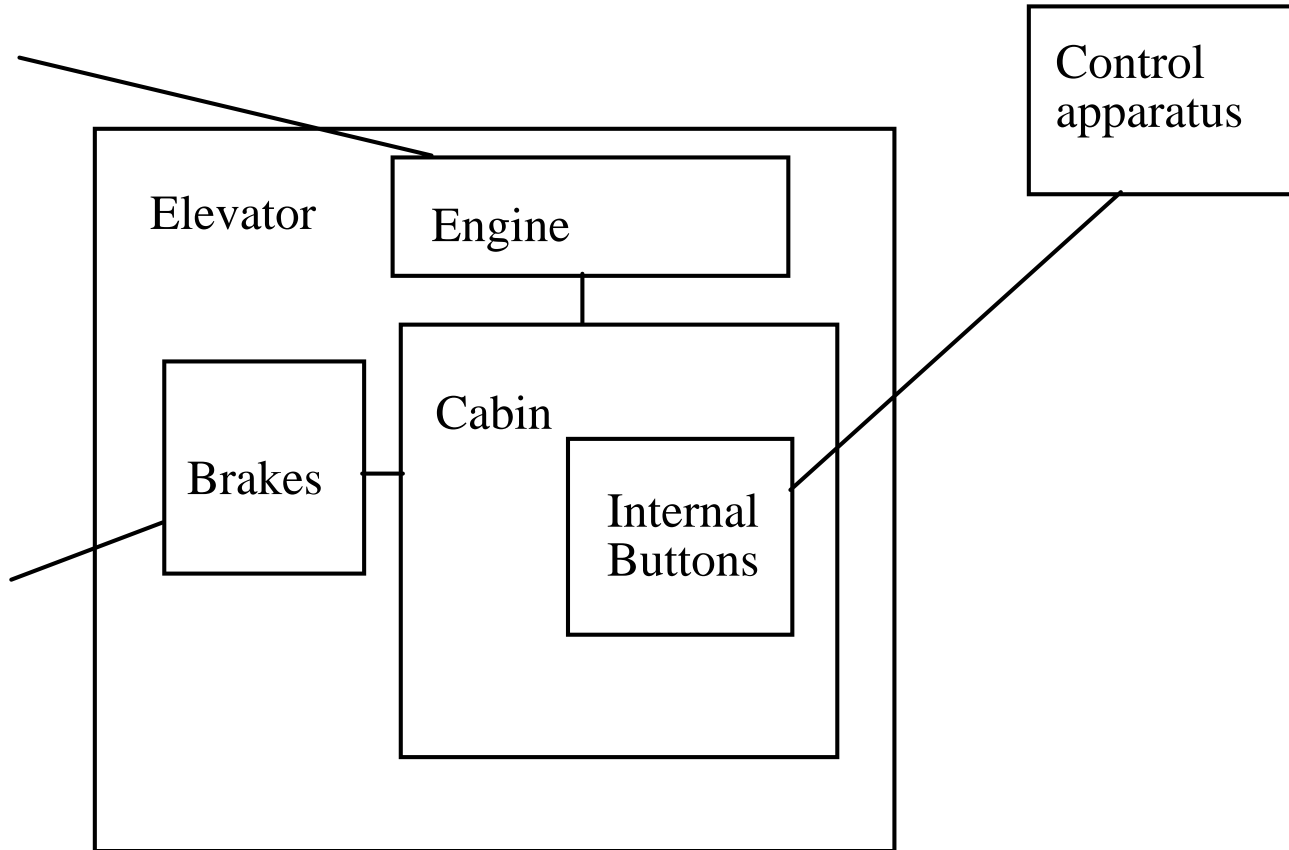
- Try to separate
  - safety
  - performance
  - usability (e.g, button illumination)
  - cost
- although some are strongly related
  - cost reduction by using cheap material can make solution unsafe

A modular structure



buttons at floor i

Module decomposition may be iterated



Abstraction

- The modular view we provided does not specify the behavior of the mechanical and electrical components
  - they are abstracted away

Anticipation of change, generality

- **Make the project parametric wrt the number of elevators (and floor buttons)**