Large-Scale Application Development and Integration

Large Scale Systems
System Decomposition

Course notes compiled from:
Jim Fawcett
Dave Penny
Software Complexity

• “Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one, a subroutine” … “In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.”

• “Digital computers are themselves more complex than most things people build. They have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders of magnitude more states than computers do.”

• “The complexity of software is an essential property, not an accidental one.”

• “Many of the classical problems of developing soft-ware products derive from this essential complexity and its nonlinear increases with size.”
  - “…difficulty of communication among team members which leads to product flaws, cost overruns, schedule delays.”
  - “…difficulty of enumerating, much less understanding all the possible states of the program, and from that comes the unreliability.”

Frederick P. Brooks, Jr., “The Mythical Man-Month”
Software Development Effort

- Fred Brooks, Program Manager for IBM System/360 and Operating System/360, and later Chair of Computer Science Department, UNC at Chapel Hill, uses this example in his famous “Mythical Man Month”.

- Most of us conceive of a program as implementing a specific function, running on the platform on which it was developed. Customers want a program product and often a program system product. Note that there is, by Brooks’ estimate, a nine to one factor in effort required to build the later over the former.

- I develop reasonably well tested code at the rate of about 300 lines per day. So for system products I should expect to generate only about 33 lines of code per day - perhaps 2/3 of that in a team environment!
Large Systems

• Example: two very large software systems successfully implemented:

  - Over The Horizon Radar (OTHR) contained about 3,000,000 lines of code, written mostly in FORTRAN with some C and some assembly language as well.

  - BSY-2 submarine battle management system software is several times larger than OTHR, written mostly in Ada.

• A 5 million line system would require something like:

\[
\frac{5,000,000 \text{ lines}}{22 \text{ lines/day} \times 240 \text{ days/year}} = 947 \text{ person years of effort}
\]

• So, to complete the project in 2.5 years would require the services of at least 380 well trained software developers.

• Two points are almost self evident:

  - a system this large must be partitioned into many relatively small, nearly independent components in order to get it to work

  - it would be much better not to create such a large system at all, but rather, to build most of it from reused sw components, reserving new code for new requirements.
Software Development Process Models: Necessity of Models

- Large scale software is developed by teams of, possibly, several hundred developers.
- Coordinating the activities of these teams requires a detailed model of the process which everyone understands.
- The model helps us ensure that each team’s products integrate smoothly with the products of all the other teams.
- Software models recognize the phases:
  - Requirements Analysis
  - Design
  - Implementation
  - Test
Development Process

• **Roles and activities for each development phase**
  - Roles are models for behavior and activities of people at various levels in a development project.
    - Project Manager
    - SW Project Manager
    - Architect
    - Team Leaders
    - Team Members
    - Support Personnel

• **Products of each development phase**
  - products are those components of the software product created at each phase of the development process
Software Product Model

- **Architecture, OCD**
  - defines product components and allocates processing to them
  - defines external product behavior
- **Requirements Specification, B-Spec, SRS**
  - describes what constitutes correct operation
  - it is the basis for testing and evaluation
- **Product Specification, C-Spec, SDD**
  - defines an architecture for the system
  - describes software design and implementation
  - specifies a software build process
- **Test Plan**
  - defines procedures for unit, integration, validation, qualification, and regression testing
  - qualification test procedures are emphasized
- **Prototype Code**
  - verifies design for critical processing, analyzes implementation problems as they arise
- **Product Code**
  - Code for each component of the product, implemented as software modules.
  - test stub attached to each module, used to establish basic software cycling and nearly correct operation
- **Test Code**
  - test drivers for unit, integration, and qualification tests
- **Test Report**
Software Architecture

• The architecture:
  - defines a partition of the product into component parts
  - describes, in detail, the product’s public interface
  - lists critical processing
  - may define major data structures

• Purpose of the Architecture
  - provides a bridge from requirements to product
  - it is the basis for estimating development costs and making schedules
  - provides an abstract model of the product to guide development of requirements, design, and implementation
  - guides the top level flow of requirements down to components and development of test plans.
  - The architecture provides a foundation, or skeleton, for the product. Developing a sound architecture is the single most important part of product development.
Architecture (cont.)

• Life Cycle
  - An architecture is developed during requirements analysis.
  - It guides top level design and evolves with the design.
  - Should be fairly static during implementation and testing.
Requirements Analysis

- Software requirements analysis and preliminary design are processes of breaking down or decomposition in the application domain:

  - Application requirements are decomposed to processes and data flows.

  - Process is a logical model of some part of the program’s activities necessary to satisfy part of its requirements model.

  - Data flows represent the information necessary to sustain activities allocated to the process.

  - Each process is allocated part of the program’s requirements model and may derive additional requirements necessary to complete or disambiguate its processing model.
Requirements Analysis (cont.)

- A design structure is developed by associating major processes with modules.

- Each such process and its data flows represents the public interface of its module.

- Each stage of the decomposition needs to flow down, or allocate, requirements to its component parts, otherwise there is no basis for deciding the correctness of the design.
Products of Requirements analysis

The main activity of the requirements analysis phase is building B-level specifications from an A-level specification.

- A-Level Specifications
  - Customer’s Requirements Specification
  - Engineering Specifications

- B-Level Specifications
  - Developer’s Requirements Specification
    - B Spec
    - Software Requirements Specification (SRS)

At the end of this phase the developers and customer hold detailed specification reviews to ensure that what is built satisfies the customer’s needs.
The Prime Directive

Partition software so that:

- each component is **cohesive** - does only one operation
- each component has **narrow coupling** with other components
- each component has **low complexity**
- each component can be nearly **exhaustively tested**
- each component is **easily understood**
- correct operation is based on satisfaction of, at most, a **few assertions**
Prime Directive

Keep it Short and Simple - the KISS principle

Applying the KISS principle is the most important step in developing correct components.
Applying the Prime Directive

• We often deal with very large, complex systems in our professional careers. How do we apply the KISS Principle?

• Divide and Conquer!
  - Partition into an executive an a set of server modules.
  - Each server is focused on a single activity.
  - Higher level modules can use the services of lower level modules.
  - Higher level modules implement the required behavior of the system and so are not likely to be reusable. They are application artifacts.
  - Lower level modules implement solution-side functionality and can be widely reused when we design with foresight.
Structured Design

• Early work of software design (from 1979) that presented concepts such as cohesion, coupling, and encapsulation.
  – “Fundamentals of a Discipline of Computer Program and Systems Design”
    • by Edward Yourdon and Larry Constantine

• Modules are not the same as for Parnas:
  – Module: A lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.
    • A function, a procedure, a method

• Normal and pathological connections between modules:

```
  normal          pathological
```

Diagram: Two modules connected by arrows, indicating normal and pathological connections.
Human limitations on dealing with complexity

- George Miller: *The Magical Number Seven, Plus or Minus Two*
  - Can’t keep track of too many things at the same time
  - Yourdon: Maximum number of subroutines called by a routine should be 5-9.
Two kinds of complexity

- Intra-module complexity
  - Complexity within one module

- Inter-module complexity
  - Complexity of modules interacting with one another

Total errors is a combination

Inter-module effect grows as the number of modules grow

Intra-module effect decrease as the modules become smaller

# of modules
Overall cost

- The overall cost of a system depends on both:
  - The cost of production (and debugging)
  - And the cost of maintenance
    - Both are approximately equal for a typical system

- These costs are directly related to the complexity of the code
  - Complexity injects more errors and makes them harder to fix
  - Complexity requires more changes and makes them harder to effect

- Complexity can be reduced by breaking the problem into smaller pieces
  - (So long as the pieces are relatively independent of one another)

- But eventually the process of breaking pieces into smaller pieces creates more complexity than it eliminates.
  - 1970’s: Happens later than most designers would like to believe
  - 2000’s: Happens sooner than most designers would like to believe
In case you don’t believe it...
Design approach

• Therefore, there is some optimal level of subdivision that minimizes complexity
  – But to reach it you need your judgment

• Once you know the right level, the key decision is to choose **how** to divide:
  – Minimize **coupling** between modules
    • Reduces complexity of interaction
  – Maximize **cohesion** within modules
    • Keeps changes from propagating
  – Duals of one another
Coupling

- Two modules are **independent** if each can function completely without the presence of the other
  - They are decoupled, or uncoupled

- Highly coupled modules are joined by many interconnections and dependencies
  - And loosely coupled modules have a few interconnections and dependencies

- **Goal:** Minimize coupling between modules in a system
  - Coupling translates into “the probability that in coding/modifying/debugging module A we will have to take into account something from module B”

- Note that a system that has only one module (function) is absolutely uncoupled
  - But that’s not what we want!
  - (We’ll analyze *cohesion*, coupling’s complement, later)
Influences on coupling

• Type of connection
  – Minimally connected: parameters to a subroutine
  – Pathologically connected: non-parameter data references

• Interface complexity
  – Number of parameters/returns
  – Difficulty of usage

• Information flow
  – Data flow: Passing data is handled uniformly
  – Control flow: Passing of flags governs how data is processed

• Binding time
  – More static = more complex
    • E.g., literal “30” vs. pervasive constant N_STUDENTS, vs. execution-time parameter
Common-environment coupling

- A module writes into global data
- A different module reads from it (data or, worse, control)
Cohesion

• While minimizing coupling, we must also maximize cohesion
  – How well a particular module “holds together”
    • The cement that holds a module together
  – Answer the questions:
    • Does this make sense as a distinct module?
    • Do these things belong together?

• Best cohesion is when it comes from the problem space, not the solution space
  – Echoed years later in OOA/OOD
Levels of lack of cohesion (roughly from worst to best)

- **Coincidental**
  - No reason for doing two things in the same routine
    - `double computeAndRead(double x, char c);`

- **Logical**
  - Similar class of things that still should be separated
    - `char input(bool fromFile, bool fromStdIn);`

- **Temporal**
  - The fact that things happen one after the other is no excuse to put them in the same routine
    - `void initSimulationAndPrepareFirst();`

- **Procedural**
  - Operations are together because they are in the same loop or decision process, but no higher cohesion exists
    - `typeDecide(m); // Decide type of plant being simulated and perform simulation part 1`
Levels of lack of cohesion
(roughly from worst to best) (cont)

- **Communicational**
  - Procedures that access the same data are kept together
    - void printReports(data x); // Outputs day report and monthly summary

- **Sequential**
  - A sequence of steps that take the output from the previous step as input for the next step
    - string compile(String program) {parse, semantic analysis, code generation}

- **Functional**
  - That which is none of the above
  - Does one and only one conceptual thing
  - Equivalent to information hiding
    - double sqrt(double x);
Practical Issues with Modularity

• Subdividing code
• Interface specification given modules
• Modular component design and reUse
Modularity

- The purpose of a module or class is to implement a small, simple logical model.
- The purpose of modularization is to build a software system out of cohesive, reliable modules.
- Modularization consists of dividing a program into modules which can be compiled separately. C++ performs type checking across module boundaries.

- Modules in C# and C++ are simply separately compiled files.
- We place module interface declarations in header files.
- Module implementations are placed in separate files which include the header file at compilation time via a preprocessor `#include “mod_name.hpp”` directive.

```
// Public Interface
header file
mod_name.hpp

// Module Implementation
mod_name.cpp

// Client
Implementation
client_name.cpp

#include “mod_name.hpp”
```

```
mod_name module

compile and link these files
```
Encapsulatation and the Information Cluster

“An information cluster is a set of [functions] used for every access to data that has a complex structure, sensitive security, or device dependence.”

Information Clustering

- The major benefit of this organization is that knowledge of specific layout and implementation details is hidden from clients, who have access only to a public interface.

  The internal data could be reorganized, to improve performance say, without adversely affecting any of its clients provided that the public interface remains fixed.

- Classes are simply patterns for information clusters. Objects are their instances, defined in memory.

- Modules are information clusters with only one instance.
Objectives: Software Reuse

- Hardware reuse has been widely accomplished.
  - standard integrated circuits: multiplexers, ALUs, processors, memory...
  - Standard boards: processor boards, memory boards, interfaces for RS232, RS449, ...

- Reuse has not been nearly as successful in the software industry.
ReUse

• We reuse operating systems, code generated by development tools like LEX and YACC, and standard libraries.

• But, we develop most application code from scratch, augmented only by compiler vendor’s libraries.

• Structured and Object oriented design, using interfaces and class factories give us the tools to effectively reuse application components.

• Using object technology, we can now build objects nearly equivalent to software integrated circuits.
Objectives: Design in Application Domain

- Modules and classes model logical behavior of real world entities.
  - Modules and objects are accessible to client code through public interfaces which model logical behaviors of real world objects. The object’s internal complexity is hidden from client’s view in a private or protected implementation.

- Names and behaviors of modules and classes should be based on:
  - the application model (top level components)
  - specifics of a solution model (bottom level components)

- OOD languages, like C++ and C#, provide a lot of syntactic and functional support for creating application and solution domain models.