Software Architecture for DSD

Overview
- The effects of code structure on DSD
- The problem of distributing work
- Review: the role of architecture in determining system qualities
- Module structure: designing components and interfaces for distributed teams

Code Structure and DSD
- Problems of coordination and control are affected by the way the code is structured (decomposed into parts and the relationships between the parts)
  - Problem of distributing work: want to be able to have different sites develop code concurrently without increasing communication overhead
  - Problem of incremental development: need to coordinate development so all the pieces are developed in the right order for each increment
  - Problem of run-time dependencies: timing dependencies at run require synchronization
- Focus on systematic approaches to architectural design

Problem of Distributing Work
- What kinds of issues arise when we want to have different sites develop different parts of the system?
  - Need to divide the system into parts
  - Have parts worked on concurrently by distributed teams
  - Have a working system when the parts are put together for testing or integration (each increment)
Implications of Code Structure

• In practice, there are always dependencies between system parts
  – Data dependencies (component C1 uses data produced by C2)
  – Functional dependencies (component C1 performs a function needed by C2)
  – Timing dependencies (C1 must wait for C2)
  – Resource dependencies (C1 and C2 share hardware), etc.

• Coupling
  – We characterize the set of dependencies between components C1 and C2 as their **interface**
  – Components with many dependencies are characterized as **tightly coupled**

• Component interfaces = human interfaces!
  – Where components depend on each other, developers must communicate to build components that interact correctly
  – Higher coupling => more communication to get it right

Common Problems

• Unless we are careful about managing dependencies between work assignments, tend to see coordination problems

• Common kinds of complaints:
  – Site A cannot make progress on its code until site B implements X
  – A change to code being developed at A means code developed at B stops working
  – A small change to code at site A requires large changes to code at several other sites
  – Code at A doesn’t work until code at B works, but code at B doesn’t work until code at C works, but code at C needs code at A
  – Developers at site B cannot test their code until developers at A provide critical functionality
  – A and B’s components work when tested independently but deadlock when integrated

Design Goals

• Design Problem: How to structure the software to minimize the need for coordination and communication?

• Desired Solution: divide the system into parts such that:
  – Each team knows exactly what to build
  – Each team can work on their part independently (without having to know the details of what other teams are doing)
  – Teams can work concurrently
  – A change to the implementation (insides) of one part does not affect other parts
  – The parts work together when integrated

Role of Software Architecture

• Questions about properties of system structure are questions about **software architecture**

• Review: formalize what we mean by “part” and “relationships” in terms of architectural structures

• Then consider some architectural structures important for DSD
Software Architecture

Definitions

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.”

From Software Architecture in Practice, Bass, Clements, Kazman

Examples

- Different architectural structures
  - Software components
  - Component interfaces
  - Relationships among them

- Examples

<table>
<thead>
<tr>
<th>Structure</th>
<th>Components</th>
<th>Interfaces</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls Structure</td>
<td>Programs (methods, services)</td>
<td>Program interface and parameter declarations</td>
<td>Invokes with parameters (A calls B)</td>
</tr>
<tr>
<td>Data Flow</td>
<td>Functional tasks</td>
<td>Data types or structures</td>
<td>Sends data to</td>
</tr>
<tr>
<td>Process</td>
<td>Sequential program (process, thread, task)</td>
<td>Scheduling and synchronization constraints</td>
<td>Runs concurrently with, excludes, precedes</td>
</tr>
</tbody>
</table>

Implications of the Definition

- Systems have more than one architecture
  - There is more than one useful decomposition into components and relationships
  - Each addresses different system properties or design goals
- It exists whether any thought goes into it or not
  - Decisions are necessarily made if only implicitly
  - Important control issue is who makes them and when
Fit in the Development Cycle

"...The earliest artifact that enables the priorities among competing concerns to be analyzed, and it is the artifact that manifests the concerns as system qualities."

Effects of Architectural Decisions

- What kinds of system and development properties are affected by the system structure(s)?
- System run-time properties
  - Performance, Security, Availability, Usability
- System static properties
  - Modifiability, Portability, Reusability, Testability
- Production properties? (effects on project)
  - Work Breakdown Structure, Scheduling, time to market
- Business/Organizational properties?
  - Lifespan, Versioning, Interoperability, Target market
- Not affected: functionality

Which structures should we use?

- Choice of architectural views applied reflects the choice of design goals
- Compare to architectural blueprints
  - Different blueprint for load-bearing structures, electrical, mechanical, plumbing
- Each is a view of the same building
- Different views answer different kinds of questions
  - How many electrical outlets are available in the kitchen?
  - What happens if we put a window here?

Models/Views

- Designing for particular software qualities also requires the right architectural model or “view”
  - Any model can present only a subset of system structures and properties
  - Different models allows us to answer different kinds of questions about system properties
  - Need a model that makes the properties of interest and the consequences of design choices visible to the designer, e.g.
    - Process structure for run-time property like performance
    - Module structure for development property like maintainability
Architectures of Interest

• Module structure
  – Specifies the design-time decomposition of the program into work assignments
  – Useful for: separating concerns, ease of change, work breakdown
• Uses structure
  – Specification of the inter-program dependencies
  – Useful for: sub-setting, incremental development
• Process structure
  – Decomposition of the run time structure (tasks, threads)
  – Useful for: controlling run-time dependencies, concurrent execution

Designing Architecture for Distributed Teams

DSD Architectural Design Goals

• Limit the necessity for communication by limiting the dependencies (coupling) between components
• Goals: divide the system into work assignments such that
  – Each part can be assigned to a different team and developed independently
  – Parts can be independently verified
  – It is possible to change the implementation details of one module without affecting other modules
  – Only properties of the system that are unlikely to need to be change are used by other modules

Information-Hiding Structure

• Architectural model: called the “information hiding” structure
• Components
  – Called modules
  – Leaf modules are work assignments
• Relations
  – “submodule-of”
  – The set of submodules of any module X partition X’s functionality
  – Constrained to be acyclic tree (hierarchy)
• Module interfaces
  – Modules at the leaves of the tree provide the methods implementing the system’s functionality
  – The set of methods and their behavior define the module interfaces
  – The interface methods provide the only access to a module’s internal state
  – Information encapsulated (internal) by the module are called its “secrets”
What is a module?

- **Goal:** divide the software into independent work assignments. Each work assignment is called a “module.”
- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the system
  - Its secrets: what the module hides (encapsulates). Design/implementation decisions that other parts of the system should not depend on
- Modules are abstract, design-time entities
  - Modules are “black boxes” — specifies the visible properties but not the implementation
  - May or may not directly correspond to programming components like classes/objects

A Simple Module

- A simple integer stack
- The interface specifies what a programmer needs to know to use the stack correctly, e.g.
  - `push`: push integer on stack top
  - `pop`: remove top element
  - `peek`: get value of top element
- The secrets (encapsulated) any details that might change from one implementation to another
  - Data structures, algorithms
  - Details of class/object structure
- A module spec is abstract: describes the services provided but allows many possible implementations
- Note: a real spec needs much more than this (discuss later)

Why these properties?

**Module Implementer**

- The specification tells me exactly what capabilities my module must provide to users
- I am free to implement it any way I want to
- I am free to change the implementation if needed as long as I don’t change the interface

**Module User**

- The specification tells me how to use the module’s services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same
Module Hierarchy

- For large systems, the set of modules need to be organized such that
  - We can check that all of the functional requirements have been allocated to some module of the system
  - Developers can easily find the module that provides any given capability
  - When a change is required, it is easy to determine which modules must be changed
- The module hierarchy defined by the submodule-of relation provides this architectural view

DSD Architectural Design Goals

- Limit the necessity for communication by limiting the dependencies (coupling) between components
- Goals: divide the system into work assignments such that
  - Each part can be assigned to a different team and developed independently
  - Parts can be independently verified
  - It is possible to change the implementation details of one module without affecting other modules
  - Only properties of the system that are unlikely to need to be change are used by other modules

Design Methodology

- Rationale for IH decomposition
- Evaluating the results
- Design exercise

Decomposition Strategies Differ

- How do we develop this structure so that we know the leaf modules make independent work assignments?
- Many ways to decompose hierarchically
  - Functional: each module is a function
  - Steps in processing: each module is a step in a chain of processing
  - Data: data transforming components
  - Client/server
  - Use-case driven development
- But, these result in different kinds of dependencies (strong coupling)
Submodule-of Relation

- To define the structure, need the relation and the rule for constructing the relation
- Relation: sub-module-of
- Rules
  - If a module consists of parts that can change independently, then decompose it into submodules
  - Don’t stop until each module contains only things likely to change together
  - Anything that other modules should not depend on become secrets of the module (e.g., implementation details)
  - If the module has an interface, only things not likely to change can be part of the interface

Applied Information Hiding

- The rule we just described is called the information hiding principle
- Information hiding (or encapsulation): Design principle of limiting dependencies between components by hiding information other components should not depend on
- An information hiding decomposition is one following the design principles that:
  - System details that are likely to change independently are encapsulated in different modules
  - The interface of a module reveals only those aspects considered unlikely to change

Module Hierarchy

Given a set of likely changes C1, C2, … Cn and following these rules, what happens:
- To each change?
- To things that change together?
- Change separately?

Evaluation Criteria

- Evaluation criteria follow from goals of the model: should be able to answer “yes” to the following review questions?
- Completeness
  - Is every aspect of the system the responsibility of one module?
  - Do the submodules of each module partition its secrets?
- Ease change
  - Is each likely change hidden by some module?
  - Are only aspects of the system that are very unlikely to change embedded in the module structure?
  - For each leaf module, are the module’s secrets revealed by it’s access programs?
- Usability
  - For any given change, can the appropriate module be found using the module guide
Module Decomposition

• Approach: divide the system into submodules according to the kinds of design decisions they encapsulate (secrets)
  – Design decisions that are closely related (likely to change together, high cohesion) are grouped in the same submodule
  – Design decisions that are weakly related (likely to change independently) are allocated to different modules
  – Characterize each module by its secrets (what it hides)
• Viewed top down, each module is decomposed into submodules such that
  – Each design decision allocated to the parent module is allocated to exactly one child module
  – Together the children implement all of the decisions of the parent
• Stop decomposing when each module is
  – Simple enough to be understood fully
  – Small enough that it makes sense to throw it away rather than re-do
• This is called an information-hiding decomposition

Specify the Module Interfaces

• The leaf modules in the hierarchy represent units of work
• For each leaf module, we specify
  – Services: the services the module provides that other modules can use
  – Secrets: implementation and design decisions the module must encapsulate
• We must also write a detailed interface spec. (the contract)

Architectures of Interest

• Module structure
  – Specifies the design-time decomposition of the program into work assignments
  – Components are modules
  – Relation is "implements secrets of"
  – Useful for: separating concerns, ease of change, work breakdown
• Uses structure
  – Specification of the inter-program dependencies
  – Components are programs
  – Relation is "requires the presence of"
  – Useful for: subsetting, incremental development
• Process structure
  – Decomposition of the run time structure
  – Components are processes (threads)
  – Relation is "synchronizes with"
  – Useful for: separation of concerns, concurrent execution

Questions?