Software Architecture for DSD

Overview

- The effects of code structure on DSD
- The problem of distributing work
- The role of architecture in determining system qualities
- Module structure: designing components 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 time also require communication
- Focus on systematic approaches to architectural design

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
Problem of Distributing Work

• What kinds of issues arise when we want to have different sites develop different parts of the system?
• Think about what we need to do
  – 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 (so the parts must be interdependent)

Why is this a problem?

• What does software structure have to do with communication problems?
• In practice, there are always dependencies between system parts
  – Data dependencies (component C1 uses data produced by C2)
  – Functional dependencies (component C1 performs a fn. needed by C2)
  – Timing dependencies (C1 synchronizes with/waits 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 are 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 very careful about system is divided into 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

Desired Properties

• Design Problem: How do I structure the software to minimize the need for coordination and communication?
• Desired Solution: divide the system into parts such that the following is true of the parts and their relationships
  – 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
Formalization

- Questions about properties of system structure are questions about *software architecture*
- Stop for a minute and formalize what we mean by “part” and “relationships” in terms of architectural structures
- Then consider some architectural structures that are important important for DSD

Software Architecture

Definitions

Working Definition

“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

“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.” - Bass, Clements, Kazman

• 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

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 structure depends the *specific design goals*
• Compare to architectural blueprints
  – Different blueprint for load-bearing structures, electrical, mechanical, plumbing
Each is a view of the same house

- Different views answer different kinds of questions
  - How many electrical outlets are available in the kitchen?
  - What happens if we put a window here?

- 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
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
  - Information encapsulated (internal) by the module are called its “secrets”

Module Hierarchy

Leaf Modules = Work assignments

Submodule-of relation
What is a module?

- Goal is to divide the software into independent work assignments. Each work assignment is called a “module.”
  - Concept due to David Parnas (conceptual basis for objects)
- 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 may not use and 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 (one module may be implemented by several 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

*Key idea:* the abstract interface specification defines a contract between a module’s developer and its users that allows each to proceed independently

Is a module a class/object?

- The programming language concepts of classes and objects are based on Parnas’ concept of modules
- To separate design-time concerns from coding issues, however, they are not the same thing
  - A module must be a work assignment for a team at design time, it should not determine the run-time structure of the code
  - The implementer must be free to implement with a different class structure as long as the interface capabilities are provided
  - The implementer must be free to make changes as long as the interface does not change
- In simple cases, we will often implement each module as a class/object
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

Design Methodology

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

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

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 calls 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

Effects of Changes

- Consider what happens to communication among teams.
- Suppose we have groups of requirements R1 – R3:
  - R1 and R3 are related and likely to change together
  - R2 is likely to change independently
- Suppose we put R1 and R2 in the same module and assign to different teams
  - What happens when R1 changes?
  - R2?
- Suppose R1 and R3 are put in the same module?

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 its 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)

An Example: The FWS Module Structure
Floating Weather Stations (FWS)

Floating weather stations (FWS) are buoys that float at sea and that are equipped with sensors to monitor weather conditions. Each FWS has an on-board computer that maintains a history of recent weather data. At regular intervals the buoy transmits the weather data using a radio transmitter.

The initial prototype for the buoy will measure the wind speed in knots. The buoys will use four small wind speed sensors (anemometers): two high-resolution sensors and two, less expensive, low-resolution sensors.

Accuracy is software enhanced by computing a weighted-average of the sensor readings over time. Each sensor is read once every second with the readings averaged over four readings before being transmitted. The calculated wind speed is transmitted every two seconds.

Over the course of development and in coming versions, we anticipate that the hardware and software will be routinely upgraded including adding additional types of sensors (e.g. wave height, water temperature, wind direction, air temperature). A system that can be rapidly revised to accommodate new features is required.

FWS Likely Changes

Likely changes
behavior
C1. The formula used for computing wind speed from the sensor readings may vary. In particular, the weights used for the high resolution and low resolution sensors may vary, and the number of readings of each sensor used (the history of the sensor) may vary.
C2. The format of the messages that an FWS sends may vary.
C3. The transmission period of messages from the FWS may vary.
C4. The rate at which sensors are scanned may vary.

Devices
C4. The number and types of wind speed sensors on a FWS may vary.
C5. The resolution of the wind speed sensors may vary.
C6. The wind speed sensor hardware on a FWS may vary.
C7. The transmitter hardware on a FWS may vary.
C8. The method used by sensors to indicate their reliability may vary.

Classifying Changes

Three classes of change

- hardware
  - new devices
  - new computer
- required behavior
  - new functions
  - new rules of computing values
  - new timing constraints
- software decisions
  - new ways to represent data types
  - different algorithms or data structures

Top-Level Module Decomposition

- Device Interface (DI)
  - Secret = properties of physical hardware
  - Encapsulates any hardware changes
- Behavior-Hiding (BH)
  - Secret = algorithms/data addressing requirements
  - Encapsulates requirements changes
- Software Decision (SD)
  - Secret = decisions by designer
  - Encapsulates internal design decisions
DI Submodules

- Windspeed Sensor Driver
  - Service: provides access to wind speed values
  - Secrets: Anything that would change if the current wind speed sensor were replaced with another. For example, the details of data formats and how to communicate with the sensor

- Transmitter Driver
  - Service: transmit given data on request
  - Secrets: details of transmitter hardware

Module Guide

- The module structure is documented in a module guide
- Contents describe:
  - The set of modules
  - The responsibility of each module in terms of the module’s secret
  - The “submodule-of relationship”
  - The rationale for design decisions
- Document purposes
  - Orientation for new team members
  - Guide for finding the module responsible for some aspect of the system behavior
    - Where to find or put information
    - Determine where changes must occur
  - Baseline design document
  - Provides a record of design decisions (rationale)

Excerpts From The FWS Module Guide (1)

1. Behavior Hiding Modules
The behavior hiding modules include programs that need to be changed if the required outputs from a FWS and the conditions under which they are produced are changed. Its secret is when (under what conditions) to produce which outputs. Programs in the behavior hiding module use programs in the Device Interface module to produce outputs and to read inputs.

1.1 Controller
Service
Provide the main program that initializes a FWS.

Secret
How to use services provided by other modules to start and maintain the proper operation of a FWS.
2. Device Interface Modules
The device interface modules consist of those programs that need to be changed if the input from hardware devices to FWSs or the output to hardware devices from FWSs change. The secret of the device interface modules is the interfaces between FWSs and the devices that produce its inputs and that use its output.

2.1. Wind Sensor Device Driver Service
Provide access to the wind speed sensors. There may be a submodule for each sensor type.

Secret
How to communicate with, e.g., read values from, the sensor hardware.

Note
This module hides the boundary between the FWS domain and the sensors domain. The boundary is formed by an abstract interface that is a standard for all wind speed sensors. Programs in this module use the abstract interface to read the values from the sensors.

Excerpts From The FWS Module Guide (3)

3. Software Design Hiding Modules
The software design hiding modules hide software design decisions based upon programming considerations such as algorithmic efficiency. Both the secrets and the interfaces to this module are determined by software designers. Changes in these modules are more likely to be motivated by a desire to improve performance than by externally imposed changes.

3.1. Wind Speed Sensor Monitor Service
Retrieve data from the wind speed sensor(s) and deposit valid data it in the Data Banker.

Secret
How to use the services provided by other modules to obtain wind speed and store it for later retrieval.

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?