Distributed Software Development Course Review

Outline

- Characteristics and risks of DSD
- Role of software engineering
- Process: coordinating work
- Requirements: building the right system
- Architecture: structuring for DSD goals
- Quality assurance: feedback and evaluation

Working Definition

- Distributed Software Development (DSD): teams in geographically distant locations collaborate to produce the work products of a software development
  - Synchronize in phases of the life cycle
  - Collaborate on artifacts from requirements to code
  - Coordinate activities among members of distributed teams
Rationale for DSD

- Expanded pool of trained workforce
- Necessity of getting closer to customers and using locality specific expertise to customize/localize products
- National policy in some countries
- Difference in development costs
- Promise of round-the-clock development that could lead to shorter intervals

Benefits Often Come at a Cost

- With DSD benefits come increased risks compared to similar co-located developments
- Schedule delays – same work takes longer
- Higher risk of failure
- Reduced product capabilities
  - Decreased functionality, qualities
  - Doesn’t meet some customer requirements
- Increased cost
  - May cost more in spite of lower labor costs
  - Schedule delays and rework increase costs

Observed Difficulties (1)

- Nature of a software project
  - Software development produces a set of interlocking, interdependent work products
    - E.g. Requirements → Design → Code
    - Implies dependencies between tasks
    - Implies dependencies between people
- Successful development requires effective coordination between people and tasks!
  - Must coordinate work (need product A to produce product B)
  - Must coordinate schedule (must finish A before starting B)
  - Must coordinate people (person P has expertise need to produce A but is busy)
Observed Difficulties (2)

- Key property distinguishing DSD from co-located development
  
  "The key phenomenon of DSD is coordination over distance." – J. Herbsleb (2007)

- All software projects require coordination
- Suggests that coordination at a distance is different
- Managing these differences is a central issue in DSD

How is DSD Different?

- In co-located projects, people build up ways of coordinating work
  - Shared process view (implicit or explicit)
  - Common vocabulary, viewpoint
  - Clear idea of expertise, responsibility
  - Free flow of information through informal channels
  - Common language, culture, backgrounds help avoid miscommunication
  - Relatively good understanding of relationships
  - People to tasks
  - Task dependencies
  - Professional and social

DSD is Different...

- In DSD many of the mechanisms for coordinating work are absent or disrupted
  - Much less communication
  - Temporal distance
  - Socio-cultural distance, e.g., language
  - Spontaneous communication declines rapidly with distance
  - Less effective communication
  - Fewer overlapping work hours
  - Low bandwidth links (e.g., email and other asynchronous)
  - Lack of awareness
  - Lack context hence knowledge of history, relationships
  - What people are doing day to day, concerns, availability
  - Incompatibilities
  - Differs in tools, processes, work products
  - Leads to confusion, misunderstandings, inconsistencies
- Some issues are observed with even small distances (e.g., 30 meters)
Software Development Problems

- Manifests as problems in coordination and control of software development
  - Difficulty establishing requirements (eliciting, understanding, negotiating)
  - Difficulty effectively distributing work
  - Difficulty detecting and correcting conflicting assumptions
  - Difficulty detecting and correcting slips in schedule
  - Difficulty managing change (especially requirements)
  - Difficulty managing development resources (schedule, personnel, budget)
- Similar to traditional SE problems, but more so
  - Work takes longer
  - Requires more effort

The Role of Software Engineering

- Purpose of SW Engineering
  - The purpose of software engineering is to gain and maintain intellectual and managerial control over the products and processes of software development
    - "Intellectual control" means that we are able make rational technical choices based on an understanding of the downstream effects of those choices
    - Managerial control means we are able to make rational choices about development resources (budget, schedule, personnel) to deliver software on time and within budget
Application to DSD

- Risks: in DSD communication difficulties and context difference lead to coordination and control problems
- Approach: apply software engineering processes, methods, and tools to mitigate specific DSD risks
- Cross-cutting concerns
  - Provide common processes to coordinate work
  - Limit dependencies between distributed sites
  - Formalize communication channels and artifacts
  - Continuous feedback for error detection and correction

Process and Planning

Why do processes vary?

- Different processes reflect different assumptions about the developmental context and goals
  - Context: project size, complexity, availability of stakeholders
  - Goals: time-to-market, reliability, usability, maintainability, control of risk
- Process is something we can design to address project needs
- Two distinct concerns
  - What kind of process should we use to address the specific issues of DSD?
  - How detailed and formal should the process be?
Summary Co-located vs. DSD

Co-located Development
- Free flow of information through informal means
- Shared process view
- Clear idea of expertise, responsibility
- Common culture eases understanding
- Understand relationships
  - People to tasks
  - Task interdependencies

DSD Risks*
- Restricted flow of information, mostly formal
- Possibly different process views
- Unclear idea of expertise, responsibility on remote teams
- Possible misunderstandings due to cultural/language differences
- Vague or incorrect understanding of relationships

*Standardizing the process helps mitigate these risks
a people fill roles with well-defined responsibilities

What kind of process?
- Incremental development strategies provide mechanisms for addressing DSD risks

Incremental Development Over Time
- Acts as a feedback loop with a calibration point at each delivery
  - Allows cross checking of assumptions, understanding
  - Early check if remote sites are doing what is expected
  - Early check for communication effectiveness
  - Allows plan adjustments at each increment
How formal?

- Well defined process helps address risks
  - Everyone has common definition of the sequence of activities, artifacts to be produced and their dependencies
  - Assigning roles clearly defines responsibilities
  - Helps make clear what people should be working on
  - Helps make clear when a task is finished
- Should answer for individuals the questions
  - Is this my job?
  - What do I do next?
  - Am I done yet?
  - Did I do a good job?

However: not enough just to define the process, must check that people understand and follow it.

How do we define a process?

- Process: we define a process as set of artifacts, activities, roles and the relationships between them
  - Roles produce artifacts by performing activities
  - A designer produces a design document as part of creating the design
- Specify the process in these terms
- Project plan assigns
  - People to roles
  - Artifacts to deliverables
  - Activities to tasks over time

Requirements
**Requirements Phase Goals**

- What does "getting the requirements right" mean in the systems development context?
- Only three goals
  1. Understand precisely what is required of the software
  2. Communicate that understanding to all of the parties involved in the development (stakeholders)
  3. Control production to ensure the final system satisfies the requirements
- Sounds straight-forward but hard to do in practice
- Critical because early errors are most expensive, hardest to fix

---

**Inherent Difficulties**

- Comprehension (understanding)
  - People don’t (really) know what they want (…until they see it)
  - Superficial grasp is insufficient to build correct software
- Communication
  - People work best with regular structures, conceptual coherence, and visualization
  - Software's conceptual structures are complex, arbitrary, and difficult to visualize
- Control (predictability, manageability)
  - Difficult to predict which requirements will be hard to meet
  - Requirements change all the time
- Inseparable Concerns
  - Many requirements issues cannot be cleanly separated
  - E.g., performance vs. security, safety vs. time to market
  - Difficult to apply "divide and conquer".
  - Must make tradeoffs where requirements conflict: requires negotiation among stakeholders with competing interests

---

**Additional Risks of DSD**

- DSD tends to aggravate existing difficulties (as we have experienced)
- Comprehension
  - Different languages, cultures, expectations
  - Greater risk of ambiguity
- Communication
  - Limited bandwidth, language, culture, invisible stakeholders
  - More difficult to negotiate common understanding, may miss problems
- Control
  - Less feedback, often delayed
  - Easy to get out of synch and not know it
- Inseparable concerns
  - Difficulty making clear divisions, allocation of responsibility among sites
  - Conversely, easy to have inadvertent side effects on other code
Requirements Engineering Strategies

- Apply risk mitigation in the context of the project’s software process and derived plan
- Activities: deploy explicit requirements activities
  - Requirements exploration and understanding
  - Requirements negotiation (and explicit signoff)
  - Requirements specification
  - Requirements verification and validation (feedback)
- Artifacts: provide vehicles for capture, communication & assessment, e.g.,
  - Prototypes
  - Common-requirement specification
  - Reviews
- Roles: create clear responsibilities for activities, artifacts, and communication, e.g.,
  - Analyst: exploration, customer interaction, negotiation
  - Requirements Engineer: specification
  - Reviewer: verification and validation

Suggested Approach

- Plan explicit process activities, artifacts, and roles around the three goals
  1. Understand precisely what is required of the software
     - Requirements elicitation, analysis: interact with customer and other stakeholders
     - Develop prototypes, etc.
  2. Communicate that understanding to all of the parties involved in the development (stakeholders)
     - Develop documentation and other materials
     - Incorporate feedback
  3. Control production to ensure the final system satisfies the requirements
     - Verification of architecture, module specs, etc., against requirements
- For each of these goals, need to ask the fundamental questions
  - How do we address this goal in the process?
  - How do we check how well we are meeting the goal?

Software Architecture for DSD
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

Effects of Architectural Decisions (What?)

- 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, Concurrency, Project cost, time to market
- Business/Organizational properties?
  - Lifespan, Versioning, Interoperability, Target market

Which structures should we use?

- Choice of structure depends on which qualities we want to design for
- Compare to architectural blueprints
  - Different blueprint for load-bearing structures, electrical, mechanical, plumbing
  - Designing for particular qualities requires the right view
    - Process structure for run-time property like performance
    - Module structure for development property like maintainability
DSD Architectural Design Goals

- Limit the necessity for communication: decompose the system into parts such that
  - Each part can be assigned to a different team and developed independently
  - Parts can be independently verified
  - Properties of the system that are likely to change (e.g., implementation details) are encapsulated
  - Only properties of the system that are unlikely to need to be changed are part of the architecture (interfaces)
  - Role of each part in the overall system is clear (and when together, implement the requirements)
- Support incremental development
  - Can identify a subset of parts that implement a useful subset of the system
  - Can add capability by extending the working system

Architecture and Communication

Goal: distribution of work on components requiring lead inter-team communication over life cycle

- Tasks clearly defined
- Dependencies limited
- Services manifest
- Interfaces stable & change tolerant

Key Architectural Structures

- Module Structure
  - Decomposition of the system into work assignments
- Module Interface Specs
  - Abstract specification of module’s services
  - Everything a user needs to know to use the module’s services correctly
Modularization a la Parnas

- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a "module."
- A module is characterized by two things:
  - Its interface: what the module provides to other parts of the systems. Properties other parts of the system may assume and services they can use.
  - Its secrets: what the module hides (encapsulates). Properties other parts of the system may not use and should not depend on.
- Modules are abstract, design-time entities that may or may not directly correspond to programming components like classes/objects

General Approach

- General approach to module decomposition based on principle of "information hiding"
- Information hiding: design principle of limiting dependencies between components by hiding (encapsulating) information other components should not depend on
- An information hiding decomposition is one following the design principles that (Parnas):
  - 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

Create Module Interface Specifications

- Ideally, specifies all the assumptions that the developers of one module can make about another
- Clearly documents the behavior of the module
- Clearly documents the interfaces used by the module
- Improves the ability to isolate errors quickly
- Defines implementer’s work assignment
  - Interface specification is essentially a contract between the developer and users
- Enables straight-forward mapping between use case requirements and methods
  - reduces effort required for requirements traceability
Design for Extension and Contraction

The "Uses" Relation

Difficulties

- Where subsets and extensions are not planned for, likely difficult to do
  - Removing capabilities results in other components not working
  - Capabilities cannot be added without changing existing system modules (e.g., adding/changing services)
  - Extending or contracting requires redesign
- Problems follow from unplanned dependencies
  - Arise by default during development (e.g., when creating functional behavior for use cases)
  - Module developers are free to use the services of every other module
  - Little thought given to downstream implications

Uncontrolled Dependencies

- Result of unplanned development is typically a network of dependencies (undirected graph)
- When will I have a working system?
- What happens if I need to deliver a subset?
- What do I want this to look like?
Uses Hierarchy

- "Ideal" design gives "loop-free" hierarchy with uses relation (acyclic tree)
  - Level 0 uses nothing else
  - Level N only allowed to use services on N-1 (or below)
- Defines constraints on
  - Build/test order
  - Increments & subsets
  - Layers
- Key: design the uses hierarchy to support incremental development

QA

Testing
Inefficiencies in Peer Review
Active Reviews

When

- As early as possible
  - Reduce the gap between making an error and fixing it
    - Ideally to "immediate" ... which we call "prevention" or "syntactic checking", and involves deep changes in process and notations
- Throughout development
  - People make mistakes in every activity, so every work product must be tested as soon as possible
What

- Narrow view:
  - Testing is executing a program and comparing actual results to expected results
- Wider view:
  - "Testing" is shorthand for a variety of activities: anything we can do to check for defects.
  - Dynamic program testing is the most common activity when the artifact is program code
  - Other activities include reviews, analysis of models, automated checks.

Testing is executing a program and comparing actual results to expected results.

Wider view:
- "Testing" is shorthand for a variety of activities: anything we can do to check for defects.
- Dynamic program testing is the most common activity when the artifact is program code
- Other activities include reviews, analysis of models, automated checks.

We usually need several.

The Long When

- Test execution is just one part of testing
  - And it needs to be a very cheap, automated part, because we should re-test the program over and over as it evolves
- Test design can often be done much earlier
  - Example: Part of a good system design is devising acceptance test cases.
  - Test design is also a test of specifications
    - Is this specification precise, or ambiguous? Can I effectively check whether an implementation satisfies it?

Test design can often be done much earlier.

Peer Reviews

- Systematic review (by developer’s peers) with express purpose of finding defects
- Generally considered most effective manual technique for detecting defects
  - E.g., Bell-Northern found 1 hour code inspecting saves 2 to 4 hours code testing
  - Effect is magnified in requirements inspections (e.g., 30 times in one study)
- Means that you should be doing peer reviews, but...
  - Doesn’t mean that manual inspections cannot be improved
  - Doesn’t mean that manual inspections are the best way to check for every properties (e.g., completeness)
  - Should be one component of the overall V&V process

Peer Reviews
Active Review Process

- Role: Quality Assurance
- Activities
  - Identify several types of review each targeting a different type of error (e.g., UI behavior, consistency between safety assertions and functions)
  - Identify appropriate classes of reviewers for each type of review (specialists, potential users, methodology experts)
  - Assign reviews to achieve coverage: each applicable type of review is applied to each part of the specification
  - Design review questionnaires (key difference)
  - Conduct review
  - Revise
- Artifacts
  - Input: documentation to be reviewed and support docs (e.g., module spec plus requirements/use cases)
  - Output: Review questionnaires

Why Active Reviews Work

- Particularly good fit for DSD
  - Reviews can be conducted concurrently
  - Does not require big meetings
  - Forces remote reviewers to actually use the documents
  - Provides clear feedback on whether or not remote parties share understanding of the work products
- Focuses reviewer’s skills and energies where they have skills and where those skills are needed
- Reviewers must participate actively but responses are relatively private (not public criticism)
- Downside: much more work for V&V (but can be productively pursued in parallel with document creation)

Questions?