Distributed Software Development

Summary of SE Methods and Principles

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
Observed Difficulties (1)

- Nature of a software project
  - System 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
- Experience suggests that coordination at a distance is different
- Managing these differences is a central issue in DSD
Informal Communication Pathways

• In co-located projects, people build up informal 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 misunderstanding
  – 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
    • Differ in tools, processes, work products
    • Leads to confusion, misunderstandings, inconsistencies
Software Development Risks

• Manifests as problems in coordination and control of software development
  – Difficulty establishing requirements (eliciting, understanding, negotiating)
  – Difficulty detecting and correcting conflicting assumptions
  – Difficulty controlling quality
  – Difficulty managing change
  – Difficulty effectively distributing work
  – Difficulty detecting and correcting slips in schedule
  – Difficult managing development resources (schedule, personnel, budget)
• Similar to traditional SE problems, but more intense

Useful to View as Risks

• Risk of building the wrong software (intellectual)
  – Misunderstand the requirements
  – Miss requirements or fail to address them
  – Functions needed by distributed team members not implemented or implemented incorrectly
• Budget, schedule, personnel risks (managerial)
  – Balancing workload
  – Developing common understanding schedule, sequencing
• Fundamental SE issue is how to mitigate DSD risks
Purpose of SW Engineering

• Defn: 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

Example: Requirements

*Lessons Learned in Distributed Software Development* – Komi-Sirvio, & Tihinen.
Risk Mitigation Strategies

- Build risk mitigation into the project’s software process
- Requirements are missing, misunderstood
  - Requirements exploration with stakeholders (customer)
  - Early modeling: prototypes, mockups
  - Precise documentation
  - Active reviews to double check communications
  - Incremental delivery
  - Clear responsibilities for requirements tasks, products
- Requirements change
  - Must address at all stages of process
  - Consider the effects of changes in advance
  - Software design for robustness, ease of change
  - Explicit processes for managing change (baseline, change request, triage, approval, dissemination, tracking, etc.)

Process View

<table>
<thead>
<tr>
<th>Activities</th>
<th>Artifacts</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Prototypes, use cases, etc.</td>
<td>Customer</td>
</tr>
<tr>
<td>Elicitation &amp; negotiation</td>
<td>ConOps</td>
<td>Analyst</td>
</tr>
<tr>
<td>Specification</td>
<td>Formal SRS</td>
<td>Reviewer</td>
</tr>
<tr>
<td>Verification and validation</td>
<td>QA plan (reviews, tests)</td>
<td></td>
</tr>
<tr>
<td>Architecture</td>
<td>Module structure</td>
<td>Architect</td>
</tr>
<tr>
<td>Design for distribution</td>
<td>Module interface specs.</td>
<td>Reviewer</td>
</tr>
<tr>
<td>Design for change</td>
<td>Active review</td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detailed Design/code</td>
<td>Test cases</td>
<td>Coder</td>
</tr>
<tr>
<td>Verification and validation</td>
<td></td>
<td>Tester</td>
</tr>
<tr>
<td>System Integration/test</td>
<td>Test cases</td>
<td>Customer</td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change Management</td>
<td>Change Request</td>
<td></td>
</tr>
</tbody>
</table>
Planning for Distributed Teams

Risks: Coordinating Work at a Distance

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.
Risk Mitigation

- Iterative development
- Well defined process, roles
- Clear plan, work assignments
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

Well-defined Process Benefits

- Process should also be relatively formal
  - Written down in detail
  - Required for all of the distributed sites
- Well-defined process clearly specifies
  - The artifacts to be produced
  - The set of activates that must be performed
  - The set of roles
  - The relationships
    - Which roles perform which activities to produce which artifacts
    - The order of activities
    - Which artifacts are needed as input to produce which other artifacts
From Process to Plan

- Process definition manifests itself in the project plan
  - Process definition is an abstraction
  - Many possible ways of implementing the same process
- Project plan makes process concrete, it assigns
  - People to roles
  - Artifacts to deliverables
  - Activities to tasks over time
- For DSD, it is essential that distributed teams agree on the project plan

Roles and Responsibilities

- Well defined roles provide a badly needed structure
  - Define who is responsible for what
  - Gives guidance for expected expertise
- Relations between roles tell you
  - Who needs to talk to each other (e.g., shared responsibility, handoff, etc.)
  - What you need to be talking about
  - Provides bases for forming professional relationships
- Upshot: in DSD it is critical that
  1. Roles and their responsibilities are clearly defined
  2. Well defined lines of communication are established between roles at different sites
  3. People consistently perform their role’s responsibilities
Software Architecture for DSD

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 characterizes kinds different system properties
- 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

- System and development properties affected by the system structure(s)
- System run-time properties
  - Performance, Security, Availability, Reliability, Concurrency
- 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
- But not function

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 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)

**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
In practice this means…

• Want to decompose into components that
  – Are team work assignments
  – Implement all requirements
• Inter-component dependencies are minimized
  – Expose only what is necessary
  – Encapsulates properties likely to change
• Services provided by each component are well defined (precise, unambiguous, complete)

Module Structure and Interface Design

Design Principles
Considerations in Interface Design
Role of Information Hiding and Abstraction
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
  – Module interfaces specify everything one must know to use the module’s services correctly
  – Modules encapsulate properties other modules should not depend on

Module Hierarchy

Submodule-of relation
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

Three Key Design Principles

- Address the basic issue: which constructs are essential to the problem solution vs. which can change
  - “Fundamental assumptions”
  - “Likely changes”
- Most solid first
- Information hiding
- Abstraction
Principle: Most Solid First

- View design as a sequence of decisions
  - Later decisions depend on earlier
  - Early decisions harder to change
- Most solid first: in a sequence of decisions, those that are least likely to change should be made first
- Goal: reduce rework by limiting the impact of changes
- Application: used to order a sequence of design decisions
  - Generally applicable to design decisions
  - Module decomposition – ease of change
  - Developing families – create most commonality

Information Hiding

- Information hiding: 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 (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
Abstraction

- General: disassociating from specific instances to represent what the instances have in common
  - Abstraction defines a *one-to-many relationship* E.g., one type, many possible implementations
- Modular decomposition: Interface design principle of providing only essential information and suppressing unnecessary detail

Abstraction

- Two primary uses
- Reduce Complexity
  - Goal: manage complexity by reducing the amount of information that must be considered at one time
  - Approach: Separate information important to the problem at hand from that which is not
    - Abstraction suppresses or hides "irrelevant detail"
    - Examples: stacks, queues, abstract device
- Model the problem domain
  - Goal: leverage domain knowledge to simplify understanding, creating, checking designs
  - Approach: Provide components that make it easier to model a class of problems
    - May be quite general (e.g., type real, type float)
    - May be very problem specific (e.g., class automobile, book object)
Applying Principles

• Most-solid: asks the questions
  – Which design decisions are least likely to change?
  – Equivalently: would this decision be true of any implementation we will do? Would it be common to all applicable implementations?

• Information hiding
  – Are these design decisions likely to change together or independently?

• Abstraction
  – Which capabilities are essential to the problem that must be solved?
  – Which details are irrelevant to the problem?

Module Decomposition

- Identify likely changes
  - Change together
    • Tightly coupled
    • Same module
  - Change independently
    • Separate modules

Submodule-of relation
Face Recognition

• Take advantage of existing web services
• Support a family of possible applications

Decomposition Strategy

• Decompose recursively
  – If a module holds decisions that are likely to change independently, then decompose it into submodules
  – Decisions that are likely to change together are allocated to the same submodule
  – Decisions that change independently should be allocated to different submodules

• Stopping criteria
  – Each module contains only things likely to change together
  – Each module is simple enough to be understood fully, small enough that it makes sense to throw it away rather than re-do

• Design and define the module interfaces
  – 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

• Result: well-defined work assignments where dependencies are few and easily understood, by construction
Module Interface Design Goals

• General design goals addressed by module interface design
  – Support architectural goals: Independent work assignments, maintainability, understandability, testability, etc.
  – Addressed by two module interface design goals
1. Control dependencies
   – Encapsulate anything other modules should not depend on
   – Hide design decisions and requirements that might change
2. Provide necessary services
   – Provide all the capabilities needed by the module’s users
   – Provide only what is needed (complexity)
   – Provide problem appropriate abstractions
   – Provide reusable abstractions

1. Controlling Dependencies

• Addressed using the *information hiding principle*
  – Design principle: limit dependencies between modules by encapsulating information other modules should not depend on
• When thinking about what to put on the interface
  – Design the module interface to reveal only those design decisions considered unlikely to change
  – Required functionality allocated to the module and considered likely to change must be encapsulated
  – Each data structure is used in only one module
  – Data structures may be accessed by programs within the module but not those outside the module
  – Any other program must access internal data by calling access programs on the interface
• Origin of good OOD principles
2. Provide Services

- Interface provides the capabilities of the module to other modules in the system, addressed by:
  - Abstraction: provide only essential information and suppress unnecessary detail
  - Most solid: provide only information that will be true of any implementation of the module

Which Principle to Use

- Use abstraction when the issue is what should be on the interface (form and content)
  - How can I provide the necessary services without unnecessary detail?
  - How can I provide services that are easy to understand and use for the problem?
- Use information hiding when the issue is what information should not be on the interface
  - Which information should users avoid depending on?
  - What is likely to change or do I want to be able to change?
Interface Specification

Module Interface Specifications
- Documents all assumptions user’s can make about the module’s externally visible behavior (of leaf modules)
  • Access programs, events, types, undesired events
  • Design issues, assumptions
  • Concurrency and timing constraints
- Document purpose(s)
  • Provide all the information needed to write a module’s programs or use the programs on a module’s interface
  • Specify required behavior by fully specifying behavior of the module’s access programs
  • Define any constraints
  • Define any assumptions
  • Records design decisions

Architectural Concept

• Take advantage of existing web services
• Support a family of possible applications
• How should the face recognition module be designed?
Simple Face Recognition Module

Define services provided

1 Introduction

The Simple Face Recognition Module provides the minimal services needed to locate and identify faces in pictures.

This module provides an abstract face recognition service. Its secret is the specific face recognition services used. This includes the face recognition algorithm, data base structures, and whether the algorithm is implemented in hardware or software.

2 Services

2.1 Services Provided

<table>
<thead>
<tr>
<th>Service</th>
<th>Provided By</th>
<th>Tested By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Train the system to recognize an individual's face in an image.</td>
<td>train</td>
<td></td>
</tr>
<tr>
<td>2. Recognize an individual in a photo and returns the identifier associated with the individual.</td>
<td>recognize</td>
<td></td>
</tr>
</tbody>
</table>

Access Methods

2.2 Access Method

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Parameter name</th>
<th>Parameter type</th>
<th>Description</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>train</td>
<td>identifier:0</td>
<td>String</td>
<td>Individual identifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>protocol:0</td>
<td>URL</td>
<td>JPG photo link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>trained:0</td>
<td>Boolean</td>
<td>False if training failed</td>
<td></td>
</tr>
<tr>
<td>recognize</td>
<td>photo:0</td>
<td>URL</td>
<td>JPG photo link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>identifier:0</td>
<td>String</td>
<td>Individual identifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reliability:0</td>
<td>Percent</td>
<td>Certainty of recognition in percent</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>otherFace</td>
</tr>
</tbody>
</table>
Design for Extension and Contraction

The “Uses” Relation

PURPOSES OF SYSTEM SUBSETS

• Often good reasons for extending or contracting software capabilities
• Extensions
  – Planned upgrades in new versions
  – Develop system as a set of increments, each adding capability
• Contractions
  – Build to schedule, e.g., time-to-market driven
  – Provide lower cost, lower capability subset
  – Repurpose a subset of system for a related development
Difficulties

- Unplanned subsets and extensions likely difficult to do while maintaining quality
  - Removing capabilities results in other components not working
  - 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
- Control requires designing for extensibility

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?
The “Uses” Relation

- Relationship is formalized as the “uses” relation
- Definition: Program A uses program B if a correct version of B must be present and working correctly for A to work correctly (satisfy its Interface Spec)
  - Intuitively: Any system with A in it must also have B if A is to work correctly
- “uses” is defined over programs (e.g., services) but may be simplified as a relation between modules
- Often the same as “calls” but not always
  - A may call B but not use it (would work with a stub)
  - A may use B but not call it (e.g. garbage collection, fills buffer)

As Architectural View

- The “uses” structure exists whether any thought is put into it or not
- The structure affects a range of important system and development qualities
  - Ability to deliver increments
  - Ability to extend/contract capabilities to meet schedule
  - Portability (layers), abstract machines
  - Testability (incremental build/test)
- Meeting these kinds of design goals requires *purposeful design* of the “uses” structure
- The “uses” as architectural structure
  - Components: services or modules (depending on granularity)
  - Relation: “allowed to use”
  - Interfaces: where A uses B, service B provides that A requires
Uses Hierarchy

- For incremental subsets
  - First subset uses nothing else
  - Second only uses first or itself
  - Etc.
- Planning: if each is a module, then know the set of work assignments for each subset
- Key point: the “allowed to use” relation is something the architect designs (not just a view of an existing system)

Quality Assurance: Building Quality In

The role of “testing”*
Active reviews

*From Prof. Michal Young
Feedback in the Product Development Cycle

Effectiveness of Peer Reviews

- Generally considered *most effective manual technique for detecting defects*
  - Analysis of 12,000 development projects showed defect detection rate of 60-65% for formal inspection 30% for testing
  - Bell-Northern found 1 hour code inspecting saves 2 to 4 hours code testing
  - Effect is magnified in earlier inspections (e.g., 30 times for requirements 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)
Active Reviews

- In practice: an active review asks a qualified reviewer to check a specific part of a work product for specific kinds of defects by answering specific questions:
  - Ask a designer to check the functional completeness by showing the calls sequences sufficient to implement a set of use cases
  - As module interface reviewer to write test cases for access programs
- Can be applied to any kind of artifact from requirements to code
- Particularly useful for DSD
  - Provides reviews
  - Provides secondary check of communication effectiveness

Program Families and Software Product Lines

Building Sets of Systems
Inefficiencies of Sequential Development

- Hypothesis: much of software development is re-development.
  - Software inevitably exists in many versions
  - Seldom develop truly new applications
- Implication: typically much in common among our systems
  ...But very little is reused
  - Difficult to identify commonalities and differences
  - Difficult to reuse code components
  - Difficult to add desired feature to existing design
  - Difficult to adapt other work products (if they exist)
  - Generally easier to re-do than re-use
- What makes work products difficult to reuse?

Program Families

We consider a set of programs a family if they have so much in common that it pays to look at their common aspects before looking at the aspects that differentiate them.

- David L. Parnas
Family Architecture Design

Role of Architecture

• Architecture provides the basis for making and representing common design decisions for a family
  – Use architecture to instantiate common design decisions
  – Architectural representation provides the “intermediate representation” for the family
  – Instances of a family share common architecture, differ in design details
• How do we systematically construct one?
Sequential Development

- System developed through sequence of design decisions leading to product (8)
- Developing new product version requires backing up
  - Some decisions won’t apply to new version
  - Must back up to point where decisions can be re-made. (3, 5 or 6)
- How far we need to back up depends (roughly) on the order of decisions

Objectives of Family Development

- Exploit commonality when building similar systems
  - Deploy systems in multiple versions
  - Quickly build new versions of a system
  - Produce deliverable code and documentation rapidly
  - Reduce cost, improve quality
- Design products to be reused (changed, extended)
  - Easy to add new features/capabilities
  - Easy to produce different versions
- Focus on reuse conceptual structures
  - Code is not the hard part, low ROI
  - Want to reuse requirements, design, etc.
- Requires a strategic view of development (encompasses multiple developments over time)
Family Development Model

- Development of new system begins from an intermediate stage
  - Order of decisions is critical
  - Intermediate representation is important
- Branching = different decision
  - All decision above a branch are in common to the family members
- Most similarity achieved by making as many common decisions as possible before creating differences

Family Design Approach

- Conceptual approach using familiar concepts (there are others)
- Approach based on “design for change”
  - Most-solid-first: apply to common elements
  - Information hiding: apply to variations
- How is the system decomposed into parts?
  - System is decomposed into a hierarchy of information-hiding modules.
  - Structural decisions common to the family members are made first – result in overall structure and interfaces
  - Variabilities are pushed to the leaf modules and hidden when possible
Module Hierarchy

Focus on encapsulating variabilities.

Generic PL-Process View

Domain Engineering

Application Engineering

Customer Needs/ Business Goals

Commonality Analysis

Implement Production Environment

Family Production Environment

Model application

Build Family Instance

Economic Analysis

Domain Qualification

Domain Analysis

Domain Implementation

Frequent/ Low Cost

Infrequent/ Higher Cost

Customer System Requirements

Product

Process
Simplified P-L Economics

<table>
<thead>
<tr>
<th>Cumulative Cost</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With reuse
Payback Point
Without reuse

C = average cost to develop family member

Product-Line Development Over Time

---

… a result of “strategic” software engineering
Summary Rewards

- Exploit commonality between similar systems
  - Leverage existing expertise
  - Capture experience and results in concrete form for reuse
- Rewards
  - Achieve large-scale productivity gains
  - Improve time to market
  - Improve quality
  - Increase customer satisfaction
  - Respond rapidly to requirements changes (within P/L scope)
  - Reduce long term cost
  - Better use of skilled personnel (software engineers)

Summary Risks

- Increased up-front investment
- Requires expertise adapting
  - Products
  - Processes
  - Organization
- May not yield savings if…
  - Incorrect analysis
  - Inadequate design
  - Unpredictable changes
Summary

- Exploit commonality between similar systems
  - Leverage existing expertise
  - Capture experience and results in concrete form for reuse
- Unique in simultaneously improving quality while reducing cost, development time, and effort
- Best fit for product-oriented development organizations with documented successes
  - Cummins Diesel
  - Hewlett-Packard
  - Bosch
  - BMW
  - Daimler-Benz
  - Nokia, Etc.
- But increased up-front investment

Questions?