Different Types of Stakeholders View A Software Product Differently

External View: Customers, Investors, Bystanders

A Software Product consists of a lot of components of a lot of different types
Key Challenges of Software Engineering

- How to satisfy all of those stakeholders
- At acceptable cost
- In acceptable time frame
- And be sure that you have done it

Borrow the Approach to Building Other Types of Products

- Problem enunciation, understanding (requirements)
  - What is the problem to be solved?
- Solution formulation (architecture)
  - How might the problem be solved?
- Solution reduction to practice (design)
  - How will the problem actually be solved?
- Solution implementation (coding)
  - The actual solution to the problem
- Interconnections among all of these
- Evidence of consistency (analysis/testing)
- Intuition (?) about what makes them "good"
- Background in how to build them

Interconnections Among Components

- They are there to assure that the components have the right relations to each other
  - Test execution results are consistent with expectations
  - Code implements design
  - Test data really represents expected usage
  - Proofs of concepts really connect proofs to concepts
- Desired/required interconnections specified early in project
  - Define what it means for product to be "correct"
- Interconnections validated as product is built
  - Testing/analysis/verification/validation
- Stakeholders view evidence of validation of interconnections

Key Focus of Software Engineering

- How to describe such products?
  - Their component parts
  - Their relations/interconnections
- What processes should be used to build such products?
  - And assure that they are "good" at the end
- How to grow and evolve such products
  - At acceptable costs
  - And improving quality

Software Engineering = Products + Processes
How to Build Such Products?

- Such a complicated product must be complicated to build, and evaluate, and evolve
- Clearly many possible approaches
  - Many suggestions over the decades
  - Suggestions and controversy continue
  - Situations, stakeholders, and requirements dictate different approaches
- Important to be able to modify/adapt approach as needed
- Focus on PROCESS as complement to PRODUCT

Product/Process Duality

- Use one to understand the other
- What is a software product?
  - Study the process that builds it
  - It is the result of the operational steps that built it
- What does a process do?
  - Understand it by the products that it builds
  - A process is an input/output map

Process as an Input/Output Map

Yields two complementary sets of issues

- Macro view
  - What does it do/how does it behave
- Micro view
  - How does it effect its behavior

Each important to the other
**Macro Focus**

- Input Artifacts
- Resources: People, Money, Tools, Time
- Process
- Outputs Artifacts, Effects
- Other Behaviors
- Cost, Time, Errors

**Micro-Process Focus**

- Input Artifacts
- Resources: People, Money, Tools, Time
- Process
- Outputs Artifacts, Effects
- Other Behaviors
- Cost, Time, Errors

**Each Benefits the Other**

- Input Artifacts
- Resources: People, Money, Tools, Time
- Process
- Outputs Artifacts, Effects
- Other Behaviors
- Cost, Time, Errors

**Bridging Micro- and Macro-**

- Explain how behaviors are produced
- Suggest changes, predict their effects
- Validate changes before they are made

**Analogy to other disciplines**

- Economics
- Physics
  - Thermodynamics
  - Electricity
- Medicine/biology

**What we learn from the analogies**

- Macro- approach comes first
- Limited success in engineering
- Micro- approach/theory follows
- Facilitates more effective engineering
  - Improved predictability
  - Reduced uncertainty
  - Fewer surprises
Macro Process Approach

• Study a software process as a "black box"
• What is its phenomenology
  – Know it by its external behaviors
• Use this as the basis for engineering it
  – Improving it
  – Optimizing it
  – Managing it
• What behaviors to observe?
• How to measure them?
• What laws describe them?

Some Specific Macro Process Approaches

• Capability Maturity Models (CMMs)
  – CMM
  – HCMM
  – SCMM
  – Etc.
• ISO 9000
• Six Sigma
  – 3.4 DPMO (defects per million opportunities)
• TicKIT
• CMMI (Integrated CMM)

The CMMI

• Hypothesizes dozens of Key Process Areas (KPAs)
• Each incorporating a number of Key Practices
• These are the I/O dimensions to be measured
• Example KPAs
  – Process Definition
  – Project Planning
  – Requirements Management
  – Project Integration
  – Measurement and Analysis

Measurement of KPAs

• Assessed by questionnaires
  – ~100 questions
  – Followup interviews
• Human judgement is ultimate measurement
• Output dimensions
  – Predictability
  – Repeatability

CMMI Ratings

• Hypothesizes five levels of process maturity
• Each modeled by a different profile of answers to questionnaire questions
• High CMMI ratings confer organizational competitive advantage

The CMMI Attempts to Evaluate Predictability

• Highly mature processes are those that offer assurance of predictable results
• Highest levels of process maturity also demonstrably offer expectation of continuous process improvement
• Higher maturity seems easiest to attain when software development is in a restricted domain
ISO 9000
- ISO: International Standards Organization
  - Sets standards for all kinds of things
  - Mostly tangibles
  - Promotes cooperation and competition
- ISO 9000—series of standards for processes
  - Generally aimed at assuring quality
  - Similar in goals and methods to CMMI
  - But broader in scope

Six Sigma
- Aim is to achieve extremely high quality
- Less than 4 errors/defects in 1 million attempts
- Comes from statistics: In a normal distribution
  - 68.2% of points lie within one Standard Deviation (Sigma) of mean
  - 95% lie within two Sigmas; 99.6% within three
  - Almost none past 6 Sigmas
- Goal here is for the number of errors produced by a process to be less than the number of points out past 6 Sigmas in a normal distribution
- THIS RATIONALE HAS BEEN TOTALLY LOST NOW
- “Six Sigma” approaches are any that aim to result in very high quality/low error rates

Relation to Micro-Process
- The CMMI, Six Sigma, etc. offer very little guidance on how to develop superior processes
- Or on how to improve current processes
- Micro-process aims to materialize the process.
- Tangible process representation can be studied, analyzed, evaluated using computer science techniques
- Tangible processes can be used as solid bases for demonstrable improvement

Micro-Process Approach
- Processes are also Products
  - Engineer them the way we engineer software
  - Reason about them using software analysis approaches
- Most are represented by diagrams
- Should be represented in more detail
  - Material for a later course

Traditional Waterfall Model

But this is too easy
- Suggests that development never has to “go back”
  - But we know that isn’t how it works
- Others knew this long ago
  - In other domains
The Shewhart/Deming Cycle:

Each Iteration Improves Quality

W. Edwards Deming

- Father of modern manufacturing quality
  - Bell Labs in 1940s
  - Appreciated first by Japan
  - Now universally appreciated
- Popularized “Plan-Do-Check-Act”
- Credits PDCA to Walter Shewhart
  - Just the “scientific method”? (Francis Bacon in 17th Century)

Waterfall With Shewhart Cycles

Waterfall With More Complicated Shewhart Cycles

Waterfall With More Complicated Shewhart Cycles and Data Flow
Problem: Ambiguous, Unclear

- Lack control flow information
- When to do what?
- What really follows what, and when?
- Etc.

Questions Left Unanswered

- Where does output go?
- What to do when reviews fail?
- What causes this rework?
- What portion of activity should be done?
- How do we break this cycle?

Reuse Based Development

New Process
Requires data store semantics

Evolutionary prototyping

"Throwaway" prototyping

The Rational Unified Process

New semantics to show roles of agents
Boehm's Spiral Model

Different Traversals of the Spiral Model

Extreme Programming (XP)

• A reaction to
  – Excessive process detail
  – Excessive bureaucracy
  – Lack of focus on technical work
    » Especially coding
• Emphasis on rapid creation of running code
  – Fast cycle time
  – Fail-fast philosophy

Some XP Approaches

• Test-first programming
• Pair programming
• Scrum

Test-First Programming

• Test the product before you build it(!)
• What does that mean?
  – Create a detailed test plan
  – Create a running mockup
  – Test it (e.g. by letting users try it)
    – Make needed changes quickly
• Clearly much of the product is simulated
• But this avoids discovering serious mistakes late in development, after lots of time and money has been spent

Pair Programming

• Code is written in teams of two
• One person "drives" (writes the code)
  • The other "navigates" (guides, critiques, reacts)
    – But writes no code!
• Experiments show this can be more productive than having two work independently on different code
• Applies to creating other software artifacts types too
Scrum

- Divide development into 14 or 30 daily "sprints"
- Led by Scrum Master
- Sprint starts with goal-setting meeting
  - Identify main risks to be addressed
  - Establish list of tasks to be done
    - The "burndown list"
- Daily short meetings for status updates
  - Entire team present. Reports daily progress and obstacles (few minutes each, at most)
  - Management of "burndown list"
- MANY variations on this
- Very popular approach to SW development

Which approach to use?

- A major question in software engineering
- Depends upon
  - situation,
  - context,
  - stakeholders,
  - stakes,
  - domain,
  - etc.

XP Approaches Appear Better for

- Smaller projects
- Projects where the whole team can be together daily
- Projects that don’t have to work perfectly
- Projects where time-to-market is key

Traditional Approaches Seem Better For

- Projects with safety concerns
- Large projects
- Distributed projects
- Projects that don’t have stringent time-to-market constraints
- Projects with many, diverse stakeholders

Processes are like software

- Software development processes need to meet requirements
- They execute too
- And it makes sense to architect them, design them
- Evaluate them too

Summary

- Software products are
  - Large, complex, tightly interconnected
  - Built by processes
- Software processes are
  - Products too
- Processes and Products each contain the other
- Processes and Products are built out of the same sort of material
Workflow

- What we have been describing is more typically called “workflow”
  - Specification of the flow of processing tasks from one step to another
  - Specification of how artifacts flow
  - Specification of who does which steps
- Most workflow languages are visual
- Most are intended for informal discussion
  - Some are rigorous, precise, detailed
    - And executable

The Little-JIL Process Language

- Vehicle for exploring language abstractions for
  - Reasoning (rigorously defined)
  - Automation (execution semantics)
  - Understandability (visual)
- Supported by
  - Visual-JIL graphical editor
  - Juliette interpreter
- Evaluation by application to broad domains
  - A third-generation process language
  - A “work in progress”

Four parts to a Little-JIL Process

- Coordination diagram
- Artifact space
- Resource repository
- Agents

Hierarchy, Scoping, and Abstraction in Little-JIL

- Process definition is a hierarchical decomposition
- Think of steps as procedure invocations
  - They define scopes
  - Copy and restore argument semantics
- Encourages use of abstraction
  - Eg. process fragment reuse

The “Step” is the central Little-JIL abstraction

- Interface Badge
  - (parameters, resources, agent)
- Prerequisite Badge
- Postrequisite Badge
- Substep sequencing
- Artifact flows
- Exception type
- Continuation
- Handlers
An Example: Open Cry Auction

Iterative Flow Specified by four Sequencing Kinds

- Sequential
  - In order, left to right
- Parallel
  - Any order (or parallel)
- Choice
  - Choose from Agenda
  - Only one choice allowed
- Try
  - In order, left to right

Pre- and Post-requisites

- Steps guarded by (optional) pre- and post-requisites
- Are steps themselves
- Can throw exceptions
- May be executed by different agents
  - From each other
  - From the main step

Exception Handling: A Strong Feature of Little-JIL

- Steps may have one or more exception handlers
- Handlers are steps themselves
  - With parameter flow
- React to exceptions thrown in descendent steps
  - By Pre- or Post-requisites
  - Or by Agents

Four different continuations on exception handlers

- Complete
  - Handler was a "fixup"; substep is completed
- Continue
  - Handler cleaned up; parent step is completed
- Restart
  - Handler cleaned up; repeat substep (deprecated)
- Rethrow
  - Rethrow to parent step

Artifact flow

- Primarily along parent-child edges
  - As procedure invocation parameters
- Passed to exception handlers too
  - Often omitted from coordination diagrams to reduce visual clutter
- This has been shown to be inadequate
  - Artifacts also need to flow laterally
  - And subtasks need to communicate with each other

Exception Handing: A Strong Feature of Little-JIL

- Steps may have one or more exception handlers
- Handlers are steps themselves
  - With parameter flow
- React to exceptions thrown in descendent steps
  - By Pre- or Post-requisites
  - Or by Agents

Artifact flow
Channels and Lateral flow

- Channel is like a queue in some ways
- Can specify step(s) that can add artifacts
- And steps that can take them
- All artifacts must be of the same type
- Queue semantics are only ones allowed
- Generalizations of these are needed

Resources

- Entities needed in order to perform step
- Step specifies resource needed as a type
  - Perhaps with attributes, qualifiers
- Resource instances bound at runtime
- Exception when "resource unavailable"

Examples of Resources

- Access to artifacts: shared document, locks on databases
- People: various kinds with varying skills
- Tools: compilers, CASE tools
- Agents: Each step has a distinctly identified unique resource responsible for execution of the step (and all of its substeps)

Much research is needed here

May be complex relations among them

Resource Request Example

Agent: OODDesigner; expert tool: ClassDiagramEditor artifact: DiagramReposLock IdentifyRelationships SpecifyRelationships RefineRelationships

Resource request is a query on the Resource specification repository

Timing

- Step has (optional) deadline specification
- Exception when deadline exceeded
- Parent can proceed
  - Child may be unaware of this
Much research needed here: A major oversight in most process languages.

**Timing**

- Step has (optional) deadline specification
- Exception when deadline exceeded
- Parent can proceed
  - Child may be unaware of this

**Agents**

- Collection of all entities that can perform a step
  - Human or automated
- Process definition is orthogonal to assignments of agents to steps
  - Path to automation of process
- Have freedom to execute leaf steps in any way they want

**An Articulate Process Can Help Answer These Questions**

- How do we break this cycle?
- Where does output go?
- What causes this rework?
- What portion of activity should be done?
- What to do when reviews fail?

**High-Level Process**

**Trivial Example Elaboration of**

- Implementation
- Reuse_Implementation
- Custom_Implementation
- Look_for_Inheritance
- Look_for_Parameterized_Class
- Look_for_Objects_to_Delegate_to
Trivial Example Elaboration of Design Step "Requirements Rework"

Invocation of step originally defined as substep of Requirements

Waterfall Process with Rework in Little-JIL