An Overview of Software Development

Software Engineering
Computer Science 520/620
Spring 2012
Prof. Leon Osterweil

What are these circles in this Software Product?
Products in other disciplines

• Homebuilding
  – Architect sketches
  – Blueprints
  – Appliance Manuals
  – Violation reports
  – ??
Products in other disciplines

• Homebuilding
  – Architect sketches
  – Blueprints
  – Appliance Manuals
  – Violation reports
  – ??
• Legislation

Products in other disciplines

• Homebuilding
  – Architect sketches
  – Blueprints
  – Appliance Manuals
  – Violation reports
  – ??
• Legislation
  – Hearings
  – Laws
  – Bureaucracies
  – Court cases
  – ??
Some Key Features of These Products

- Problem enunciation, understanding
  - What is the problem to be solved?
- Solution formulation
  - How might the problem be solved?
- Solution reduction to practice
  - How will the problem actually be solved?
- Solution implementation
  - The actual solution to the problem

- Interconnections among all of these
- Evidence of consistency
- Intuition (?) about what makes them “good”
- Schedules and development histories
Some Key Features of These 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”
- Schedules and histories (development process)
How to Represent All of This?

- The artifacts/entities/components
  - The different types
  - Their decompositions
  - Their continued change, growth
- The relations among them
  - Desired
  - Actual
- Their time-varying nature

Software Artifacts as Instances of Types

- A requirement specification
  - A structure of instances of types
  - Probably hierarchical
  - Different types for different requirements(?)
- Functional Requirement
  - A DFG?
  - An FSA?
- Timing requirement
  - A first order logic specification?
  - A Petri Net?
- Robustness requirement
  - An FSA?

How to decide which to use?
Most of the rest of the course will examine the various key types of software product components and try to define them as (software) types

Relations

- Many software artifacts are relations
  - Nodes related to each other in various ways
  - The entities are often relations themselves
- Some relations are over types, some over instances
- They define, correctness, “well-formedness”
- Are the basis for evaluation
- Are used in evolution
Some Examples of “Relations”

• Executing this code must meet this requirement
• This code must conform to that design element
• This compiled code came from this compiler
• This design element addresses those requirements

• These lower level requirements are elaborations of these higher level requirements
• This is the date by which that test must be passed
• Component invocations conform to component abstract interface specifications
• Documentation describes the actual system
• ETC.....
Much of the rest of the course will examine the ways in which consistency is defined and determined through the use of relation specifications.
Much of the rest of the course will also be focused on the ways in which these artifacts are made and interconnected.

If the *product* is this complicated, then it probably takes a complicated *process* to make it.

How to define the process too?

---

**Simple example of how to develop SW: The Waterfall Model**

```
Requirements --> High-Level Design --> Low-Level Design --> Code --> Test
```

CS 620 Spring 2012 Univ. of Massachusetts
Copyright L. Osterweil, all rights reserved
Problems with the Waterfall

- It trivializes the process
  - No loops
  - No decomposition
- And things do not necessarily have to happen in this order either

OK, so here are some loops: Rework between phases
All right, all right, more loops

Still at Least One More Problem: Need both Control and Data Flow

- Which did we just see?
  - Could have been either (both?)
- DFGs lack control flow information
- CFGs lack data flow information
- When to do what?
- What really follows what, and when?
- Etc.
So how about this version?

Leaves Key Questions Unanswered

Where does output go?

What to do when reviews fail?

What causes this rework?

What portion of This activity should be done?

How do we break this cycle?
Iteration in process

- PERT had no loops
  - Too high level to be very useful
  - Need for more details
- Waterfall Diagram CFG showed loops with no useful semantics

What’s the Problem with Loops?

- Loops are hard to control
- What changes on each iteration
  - And what does not
- When to stop looping
  - And how?
- Familiar problems in application software
- Also present in software processes
What’s the Problem with Loops?

- Loops are hard to control
- What changes on each iteration
  - And what does not
- When to stop looping
  - And how?
- Familiar problems in application software
- Also present in software processes

Question: What are the loops for?

The Shewhart/Deming Cycle:

Act  Plan

Check  Do
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 problems have led to much better SW development process models

- Boehm’s Spiral Model
- Kruchten’s RUP (Rational Unified Process) Model
- Agile Models
  - e.g. Scrum
- Flexible Models
  - e.g. Process Programming

Boehm's Spiral Model
Focus on Risk in Software Development

• What are the risks?
  – Failing to satisfy stakeholders
  – Who are the stakeholders, what are their stakes?
    What are the costs of failing a stakeholder?
  – What are the costs of being sure you don’t fail?
• Boehm’s Spiral Model makes these issues clearer
• E.g. Spiral Model emphasis is on control of risk
  – Loop iterator and exit condition focus on this
Different Traversals of the Spiral Model to address different risks

- Application reqmts = low risk
  - Budget, schedule = high risk
- Stable appl. reqmts & budget errors = high risk
- Application reqmts = high risk
  - Budget, schedule = low risk

A Gallery of Process Diagrams

- Some use standard graphs to show new processes
- Some try new graphs to show existing processes
- Some do both
Process models

- Earliest design and test
  - Code → Fix

- The Waterfall
  - Feasibility
  - Requirement Specification
  - Architecture
  - Preliminary Design
  - Detailed Design
  - Code & Unit Test
  - Integration Testing
  - System Testing
  - No focus in risk, no risk management

More Realistic Waterfall

- Recognition of feedback loops
  - Confined to successive stages
- “Build it twice”
  - Early prototyping
  - The beginnings of risk management
Reuse Based Development

New Process
Requires data store semantics

Reduce risk by using things proven in past use

Repository

"Throwaway" prototyping

Create a history of past use as part of the process to get a firm grip in risk issues in order to control them better
Many new software process ideas

- Some add many details to abstract spiral model
- Some reject “waterfall-based” approaches
  - Too “heavyweight”
  - Is that exact sequence of steps always necessary?
  - Need for agility
- The rise of “agile methods”, “extreme programming”...
The Rational Unified Process

Use UML to define the process. This is a Message Sequence Diagram.

Roles for agents
Concurrency
Not as phase-sequential

Architect
Architectural Analysis
Architectural Design
Describe Concurrency
Describe Distribution
Review the Architecture
Architecture Reviewer

Designer
Use-Case Analysis
Subsystem Design
Use-Case Design
Class Design
Review the Design
Design Reviewer

Database Designer
Database Design

Use UML to define the process. This is a Message Sequence Diagram.
Some Extreme Programming (XP) Examples

• Test-first programming
• Pair programming
• Scrum
• Etc.

More on these later

The Scrum: No Sequential Phases

• Software development in a sequence of “sprints”
  – Usually 30 sprints
• Each sprint lasts a day
• Sprint starts with a short meeting
  – Every team member has 2-3 minutes
• Scrum starts with overall goal-setting
  – A “burndown list”
• Scrum ends with evaluation
  – And planning for next scrum
• Main goals
  – Empower the team
  – “Time boxing” to keep things from taking too long
  – Risk mitigation
Representations of Software Development Processes

• We have just seen a few attempts
  – DFGs
  – CFGs
  – UML
  – Combinations
• Could have seen FSMs, Petri Nets.
• Software processes are very complex, though
  – Require a great deal of modeling semantics
• Maybe too complex for pictures?

Need a focus on process in order to complement our coming focus on product components
Being Precise About Processes

- Processes are REAL entities
- Important to define them
  - Completely
  - Clearly
  - Precisely
- For all relevant stakeholders
  - Developers
  - Customers
  - Managers
  - Regulators
  - Etc.

Processes as Software

- Consist of:
  - Process Requirements, the basis for
    » Process design, evaluation and improvement
  - Process Specification/Modeling/Design
    » Support for conceptualization, visualization
  - Process Code
    » Provides rigor and complete details
    » For execution and tool integration
  - Process Analysis, Measurement, and Evaluation
    » Basis for....
  - Process Maintenance (Improvement)
- Develop processes using a process development process
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

As we define software products as instances of types, we will also define the processes by which they are developed and related to each other by defining the processes for doing these things
Process Representation

- Who are the stakeholder groups for process representations?
  - Developers
  - Managers
  - Customers
- For Microprocess?
  - Developers and Managers
- What representation notation?
  - Need more details and rigor
  - Can be more technical, formal
- Diagrams could be OK for customers
  - Would need to be consistent with formal notations

Use of Formalism

- Main advantages are precision and rigor
- Semantic breath of scope is possible.
- Main drawback is lack of clarity
  - At least for non-technical people
This is one possible formalism

Rigorous Notation is Preferable

- Represent (model) specific aspects of software system using established formalisms (eg. set theory, logic, algebra) to provide the semantics
- Focus on some aspect(s), ignore the others
  - To provide clear understanding of that aspect
  - Avoid clutter
  - Provide rigor and detail about modeled aspect(s)
- We have already seen example(s) of this:
  - Use of finite mathematics, logic, graph theory to provide semantics for diagrams
Process Diagrams

- Pictorial views of how the process is to be executed
- Major steps, tasks, phases indicated by boxes
- Flows from step to step indicated by arrows
  - control flow view
  - data flow view
- Sometimes timing estimates in boxes
- Sometimes execution agents indicated in boxes
- Usually hierarchical
- Usually data (ie. software artifacts) don’t get much attention

Some Process Diagrams Used

- DFGs are most common
- FSAs are used too
- Petri Nets are less common
- Multiple diagrams are used too
A Development process

Developers → Real World

Develop Spec

Develop Implementation

Real World → Users

JSD

Interview

Develop Spec

Sys_Spec_Diag

Develop Implementation

Sys_Impl_Diag + Sys_Spec_Diag

Design_Spec

Decomposition of a part of the Rqts. process

Model_System

RW_Model

Identify_Model_Process

Model_Process

Name_List

Connect

Connection_List + Model_Process_Name_List

Specify_Model_Process

Model_Process_List

Init_Sys_Spec_Diagram

RW_Model
Well-Defined Language is Better Yet

- Diagrams support clarity, good for customers, ??
  - Pictures support intuitive reasoning
  - Help identify gaps, shortcomings, weaknesses
  - Suggest truths, theorems, facts
  - But are generally based upon very weak semantics
    » Lack breadth of semantics
    » Often lack precision and detail
- Formal Languages, good for developers, ???
  - Strength is precision and rigor
  - Broad semantics are possible
  - Often feature considerable detail (that may interfere with clarity)

Programming Languages

- Procedural
- Rule-Based
- Functional
- Combinations of the above
- Etc...
HFSP: A Functional Decomposition Language

(a) JSD(Real_World | Design_Spec) =>
   (1) Develop_Spec(Real_World_Desc | System_Spec_Diagram)
   (2) Develop_Impl(System_Spec_Diagram | System_Impl_Diagram)
   (3) Where Real_World_Desc = Interview(Users, Developers, Real_World)
   (4) Design_Spec = union(System_Spec_Diagram, System_Impl_Diagram)

Second_level
(b) Develop_Spec(Real_World_Desc | System_Spec_Diagram) =>
   (1) Develop_System_Model(Real_World_Desc | Real_World_Model, Init_System_Spec_Diagram)
   (2) Develop_System_Func(Real_World_Model, Init_System_Spec_Diagram | System_Spec_Diagram)

Third_level
(c) Develop_System_Model(Real_World_Desc | Real_World_Model, Init_System_Spec_Diagram) =>
   (1) Model_Reality(Real_World_Desc | Real_World_Model)
   (2) Model_System(Real_World_Model | Init_System_Spec_Diagram)
   (4) Develop_System_Func(Real_World_Model, Init_System_Spec_Diagram | System_Spec_Diagram) =>
        (1) Define_Func(Real_World_Model, Init_System_Spec_Diagram | System_Spec_Diagram)
        (2) Define_Timing(Init_System_Spec_Diagram, System_Spec_Diagram | Timing)
    (3) Where System_Spec_Diagram = is_composed_of(Init_System_Spec_Diagram, System_Spec_Diagram)

Fourth_level
(e) Model_Reality(Real_World_Desc | Real_World_Model) =>
        (1) Identify_Entity_Action(Real_World_Desc | Entity_Action_List)
        (2) Draw_Entity_Structure(Entity_Action_List | Entity_Structure_List)
    Where Real_World_Model = is(Entity_Structure_List)

(f) Model_System(Real_World_Model | Init_System_Spec_Diagram) =>
        (1) Identify_Model_Process(Real_World_Model, Init_System_Spec_Diagram | Model_Process_List)
        (2) Connect(Real_World_Model, Model_Process_List | Connection_List)
        (3) Specify_Model_Process(Connection_List, Real_World_Model, Model_Process_List | Model_Process_List)
    (4) Where Init_System_Spec_Diagram = is(Model_Process_List)

Fourth_level
(f) Model_System(Real_World_Model | Init_System_Spec_Diagram) =>
        (1) Identify_Model_Process(Real_World_Model, Init_System_Spec_Diagram | Model_Process_List)
        (2) Connect(Real_World_Model, Model_Process_List | Connection_List)
        (3) Specify_Model_Process(Connection_List, Real_World_Model, Model_Process_List | Model_Process_List)
    (4) Where Init_System_Spec_Diagram = is(Model_Process_List)

(f) Connection = is(State_Vector) or is(Data_Stream)
More Elaboration

(b) \texttt{Develop\_Spec(Real\_World\_Desc | System\_Spec\_Diagram) =>}
(1)\texttt{Develop\_System\_Model(Real\_World\_Desc | Real\_World\_Model, Init\_System\_Spec\_Diagram)}
(2)\texttt{Develop\_System\_Func(Real\_World\_Model, Init\_System\_Spec\_Diagram | System\_Spec\_Diagram)}

(d) \texttt{Develop\_System\_Func(Real\_World\_Model, Init\_System\_Spec\_Diagram | System\_Spec\_Diagram) =>}
(1)\texttt{Define\_Func(Real\_World\_Model, Init\_System\_Spec\_Diagram | System\_Function, Function\_Process)}
(2)\texttt{Define\_Timing(Init\_System\_Spec\_Diagram, System\_Function | Timing)}
(3)\texttt{Where System\_Spec\_Diagram = is\_composed\_of(Init\_System\_Spec\_Diagram, System\_Function, Function\_Process, Timing)}

(a) \texttt{BOOD(Req\_Spec | Design\_Spec) =>}
(1)\texttt{Identify\_Object(Req\_Spec | Objects, States)}
(2)\texttt{Identify\_Operations(Req\_Spec, Objects, States | Operation)}
(3)\texttt{Establish\_Visibility(Req\_Spec, Objects, States, Operation | Visibility)}
(4)\texttt{Establish\_Interface(Visibility, Objects, States, Operation | Interface)}
(5)\texttt{Establish\_Implementation(Interface, Implementation)}
(6)\texttt{Where Design\_Spec = is\_composed\_of(Interface, Implementation)};

Second Level
(b) \texttt{Identify\_Object(Req\_Spec | Objects, States) =>}
(1)\texttt{Identify\_Nouns(Req\_Spec | Nouns)}
(2)\texttt{Identify\_Concrete\_Object(Req\_Spec, Nouns | Concrete\_Object)}
(3)\texttt{Identify\_Abstract\_Object(Req\_Spec, Nouns | Abstract\_Object)}
(4)\texttt{Identify\_Server(Req\_Spec, Nouns | Server)}
(5)\texttt{Identify\_Agent(Req\_Spec, Nouns | Agent)}
(6)\texttt{Identify\_Actor(Req\_Spec, Nouns | Actor)}
(7)\texttt{Identify\_Class(Req\_Spec, Agent, Server, Actor, Concrete\_Object, Abstract\_Object | Class)}
(8)\texttt{Identify\_Attributes(Objects | States)}
(9)\texttt{Where Objects = union(Concrete\_Object, Abstract\_Object, Class, Agent, Actor, Server)}

(c) \texttt{Identify\_Operation(Req\_Spec, Object, States | Operation) =>}
(1)\texttt{Identify\_Suffered(Req\_Spec, Object, States | Operation\_Suffered)}
(2)\texttt{Identify\_Required(Req\_Spec, Object, States | Operation\_Required)}
(3)\texttt{Define\_Time\_Order(Req\_Spec, Operation | Time\_Order)}
(4)\texttt{Define\_Space(Req\_Spec, Operations | Space)}
(5)\texttt{Where Operation = union(Operation\_Suffered, Operation\_Required)}

(d) \texttt{Establish\_Visibility(Req\_Spec, States, Operation | Visibility) =>}
(1)\texttt{Specify\_Objects\_See(Objects | Objects\_See)}
(2)\texttt{Specify\_Object\_Seen(Objects | Object\_Seen)}
(3)\texttt{Where Visibility = union(Objects\_See, Object\_Seen)}

(e) \texttt{Establish\_Interface(Visibility, Object, States, Operations | Interface) =>}
(1)\texttt{Derive\_Module(Object | Module)}
(2)\texttt{Specify\_Attr(States, Module | Attributes)}
(3)\texttt{Specify\_Proc(Operations, Module | Procedures)}
(4)\texttt{Specify\_Visibility(Visibility, Module | Visibility\_Spec)}
(5)\texttt{Where Subsystem = is\_in\_terms\_of(Module)}
(6)\texttt{Interface = is\_composed\_of(Attributes, Procedure, Visibility\_Spec)}
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
Introduction to Little-JIL
A Process Definition Language

- Little-JIL is
  - Broad
  - Precise
  - Detailed
  - Clear (has a visual representation)
- Well-defined by finite state machine semantics
- Executable
- Incorporates powerful semantics
  - Abstraction
  - Channels
  - Preemption
  - Etc.
Need a language as a vehicle to explore and evaluate this approach

- Precise tool for enunciating process details
- Desiderata
  - Precise
  - Detailed
  - Broad in scope
  - Clear
- Diagrams and pictures won’t do
- Workflow languages fall short

Process definition languages are hard: Must address many issues

- Blending proactive and reactive control
- Coordinating human and automated agents
  - Without favoring either
- Dealing with exceptions
- Specification of resources
- Concurrency
- Real time specification
- Assignment of agents
- Scaling
- Reuse (e.g. through abstraction)
- Preemption/abortion
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
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
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

Software engineers know this.
Less clear in workflow languages

The “Step” is the central Little-JIL abstraction

- Interface Badge
  (parameters, resources, agent)

- Prerequisite Badge
   ▼ TheStepName ▲

- Postrequisite Badge

- Substep sequencing

- Artifact flows

- Handlers

- Exception type

- continuation
An Example: Open Cry Auction

Proactive 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
Proactive 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

These step kinds support human flexibility in process performance.

Iteration usually through recursion
Alternation using pre/post requisites
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 Special Focus 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
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
- Generalizations 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)

May be complex relations among them

Resource Request Example

Agent: OODDesigner; expert
tool: ClassDiagramEditor
artifact: DiagramReposLock

Resource request is a query on the Resource specification repository
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

Try and Choice Step Kinds support human (agent) flexibility

- Implement
- Reuse_Implementation
- Custom_Implementation
- Look_for_Inheritance
- Look_for_Parameterized_Class
- Look_for_Objects_to_Delegate_to
Timing

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

Preemption Semantics

- Need to allow one step to terminate execution of another step
  - Terminated step must allow this
- Some variants of this
  - Abort a step
  - Suspend a step
  - Rollback, compensate, etc.
Preemption: One step may need to kill another

A step can be defined to be Preemptable
It is willing to receive a Preempt command from another step

Preemption Semantics

- Need to allow one step to terminate execution of another step
  - Terminated step must allow this
- Some variants of this
  - Abort a step
  - Suspend a step
  - Rollback, compensate, etc.
- Only abort is implemented now
Can This Articulate Process Definition Approach Help Answer These Questions

Where does output go?

What to do when reviews fail?

What causes this rework?

What portion of This activity should be done?

How do we break this cycle?

A better basis for proceeding
A better basis for proceeding

How to do this?

A better basis for proceeding

How to do these?
A better basis for proceeding

And how to do these too?

Trivial Example Elaboration of Requirements Step

- Develop Right Element
- Declare and Define Right
- Inter-ranks consistency check
- Requirements review
- Requirements

* rmtnreview:failed
A Part of Boehm’s Spiral Model

Scrum Process in Little-JIL
A Scrum Sprint

Some Observations

- Process engineering is important, feasible
- Effective process languages are possible
  - Borrowing from programming languages helps
    » Abstraction, scoping, exception management, concurrency, etc.
  - Transactions and Real-time are needed too
- Analysis is feasible for detecting defects
  - Basis for systematic process improvement
- Process guided execution has value
  - Needs process guided user interface management
Our Approach

- What is the goal/role of each component type?
- What is the nature of it?
  - Eg. what internal structure does it have?
- What sorts of stakeholders are interested in it?
- What sorts of questions do they generally have about it?
- What sorts of relations must it participate in?
  - Internal relations
  - External relations
- What sorts of processes deal with it?

Types of Software Product Components

- Specification of customer/user needs/desires
  REQUIREMENTS
- Specification of potential solution or solution approach
  ARCHITECTURE
- Reduction of solution approach to practice
  DESIGN
- Solution
  IMPLEMENTATION
- Evaluation of solution
  TEST PLAN
  ANALYSIS/TEST RESULTS
Types of Software Product Components

- Specification of customer/user needs/desires
  REQUIREMENTS
- Specification of potential solution or solution approach
  ARCHITECTURE
- Reduction of solution approach to practice
  DESIGN
- Solution
  IMPLEMENTATION
- Evaluation of solution
  TEST PLAN
  ANALYSIS/TEST RESULTS

Well-Integrated, consistent, correctly related to each other