Design Intent in an Agile Context
Part I

Paul S Grisham

grisham@mail.utexas.edu

Feb. 16, 2006

Goals of this Lecture

- ➔ Provide an overview of the Agile software development philosophy
- ➔ Provide an overview of Extreme Programming (XP)
- ➔ Consider the role of process, organization, and artifacts in software development
- ➔ Explore opportunities to capture and utilize design intent and design rationale
- ➔ Enumerate design intent strategies for Agile software development
- ➔ Introduce the context for the lectures for the remainder of the semester

Definitions of Intent

- Functional Intent – WHAT
  - Functional Requirements
- Design Intent – HOW
  - Scenarios / Use-cases
  - Contracts
  - Obligations
- Design Rationale – WHY
  - Criteria
  - Plans
  - Alternatives
  - Non-functional Requirements (?)

How Do We Use Intent?

- Replication
  - Use existing patterns and processes to build something new
    - Strategies, Patterns and Idioms
  - Be sure we are replicating the important things
    - Cutting off the end of the ham
- Reuse
  - Include legacy modules in new systems
    - Identify opportunities for reuse
    - Make sure we use those modules correctly
    - Identify assumptions about usage
- Modification
  - Perform risk analysis
    - Explore semantic and operational dependencies
- Maintenance
  - Identify out-of-date or invalidated assumptions

Outline of this Lecture

- ➔ Quick Review
- ➔ Agile Software Development
- ➔ Extreme Programming (XP)
- ➔ Test-Driven Development
- ➔ Communication and Documentation
- ➔ Iterative and Adaptive Design
- ➔ Agile Maintenance
- ➔ Concepts for Intent-Driven Development in an Agile context

What is Agile Software Development?

- ➔ A combination of old and new ideas to respond to:
  - ➩ Customer Needs
  - ➩ Changing Requirements (See above)

- ➔ "Agile Software Development" includes several techniques that feature:
  - ➩ Close collaboration between technical and business staff
  - ➩ Face-to-face interactions
  - ➩ Frequent demonstration of working functionality
  - ➩ Frequent delivery of business value
  - ➩ Self-organizing teams
  - ➩ Commitment to innovative craftsmanship

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

http://agilemanifesto.org/

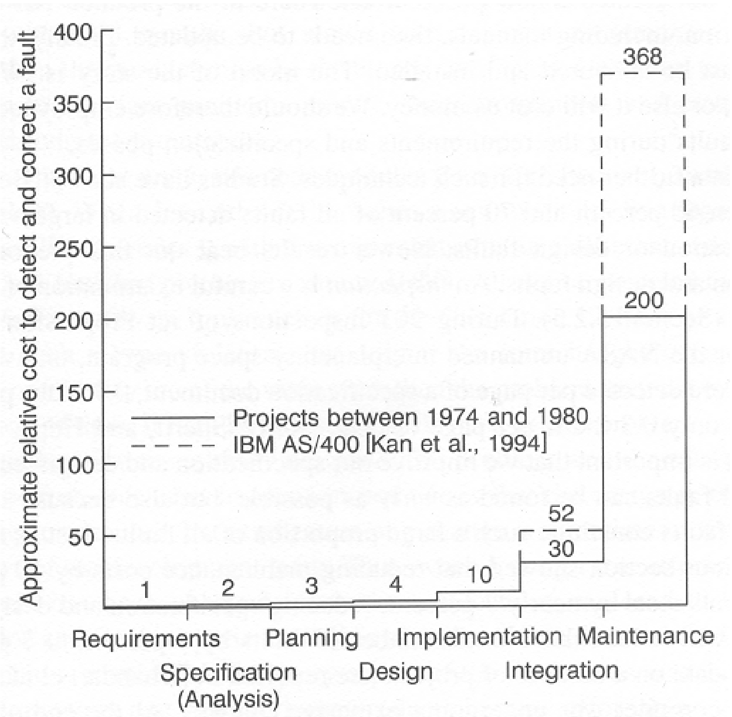
Characteristics of Plan-Driven Development

- Goal: Make process and product predictable
- Detailed planning and predictive models
  - Do we have reliably predictive models in Software?
  - Can a customer understand a software system from a model?
- Inspired by engineering methods in other fields
  - Physical engineering disciplines have well-understood functional concepts
    - A bridge is a bridge. What is a software bridge?
  - Engineering models are generally comprehensible
  - Environmental requirements change slowly
- Assume separation between design and construction
  - Design is creative and risky
  - Construction is predictable and repeatable
  - Treat programming as construction

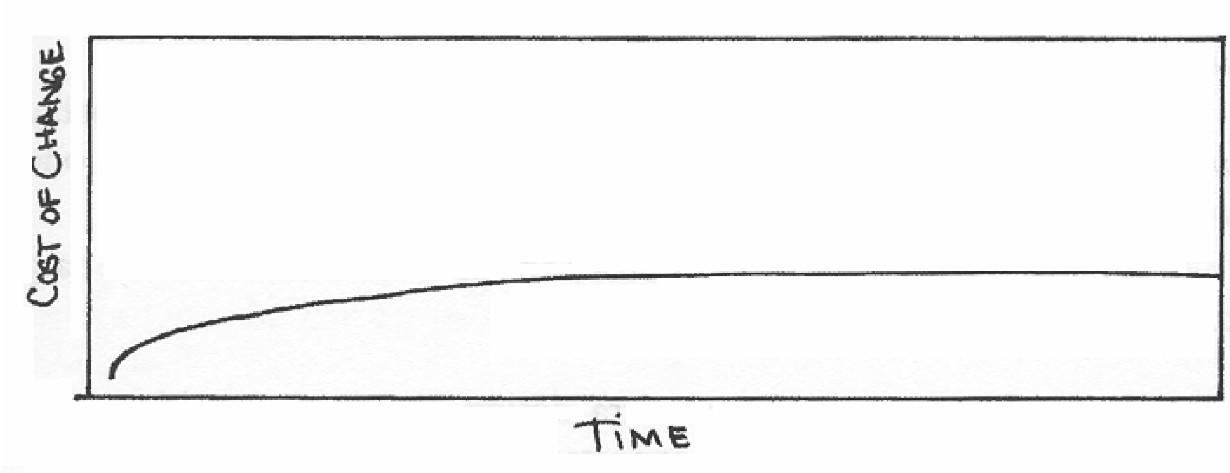
http://www.martinfowler.com/articles/newMethodology.html

The Cost of Change

- ➔ Conventional Wisdom:
  - ↳ The cost of correcting a requirements fault increases exponentially over time
- ➔ What drives these costs?
  - ↳ Inflexible designs?
  - ↳ Antiquated programming techniques? (No silver bullet!)
  - ↳ Cost to iterate the waterfall? (Process activities)
- ➔ How are they measured?
  - ↳ Compared to original cost to implement?
  - ↳ What counts as a requirements fault?



What If?



- ➔ Cost of adding a feature did not increase significantly over time?
- ➔ Requirements could be iterative and feature-driven
- ➔ We could defer long-range design decisions and focus on delivering functionality now
- ➔ Can we separate essential complexity from process complexity?

Sources of Change

- Requirements Uncertainty (Nidumolu, 1996)
  - Instability – Changes over the life of the project
    - Environmental changes – business drivers
    - Technology changes
    - Scope changes
  - Diversity – Differences between stakeholders
  - Analyzability – How requirements can be reduced to objective specifications
- Requirements errors that cause faults (Perry, ESEC93)
  - Incomplete / Omitted Requirements
  - Ambiguous Requirements
  - Lack of Knowledge
- Additional Factors
  - Emergent dependencies
  - Experimental / Conditional Requirements
  - Brooks's "Plan to throw one away"

Assumptions About Uncertainty

- Planned Development assumes requirements are:
  - Correct
  - Complete
  - Consistent
  - Analyzable – For correctness, consistency, completeness
  - Understood – No ambiguities
  - Static – Unchanging
- None of these things are true…ever
- Solutions:
  - Disallow change – Build the system as contracted, even if it no longer meets the customer's needs
  - Allow change with cost – Iterate lifecycle with impact analysis and forward and reverse traceability (cost? effort?)
  - Adopt adaptive techniques which facilitate change through goal prioritization and communication

Responding to Change

- Requirements Instability
  - Problem: Requirements change before system is deployed
  - Solution: Deploy system quickly to maximize business value
  - Solution: Evaluate which dimensions are fluid
  - Solution: Plan for the unforeseen
- Requirements Diversity
  - Problem: Different users have competing requirements
  - Solution: Identify users/stakeholders
  - Solution: Stakeholder/Viewpoint analysis
  - Solution: Negotiate and explicit documentation
- Requirements Analyzability
  - Problem: Some requirements are hard to quantify
  - Solution: ???

Responding to Change (2)

- Incomplete/Omitted Requirements
  - Solution: Rigorous interview process
  - Solution: Use of domain models to identify incompleteness
- Ambiguous Requirements
  - Solution: Formal requirements models and analysis
  - Solution: Stakeholder sign-off on requirements models
- Lack of Knowledge
  - Solution: ???
  - How do you ask what you don't know?
- Emergent Dependencies
  - e.g., Requirements inherited from choice of tech. solutions
- Experimental / Conditional Requirements
  - Requirements which depend on how solutions work in practice
  - e.g., Cumulative performance measures

Characteristics of Agile Development

- Agile Methodologies
  - Increase adaptability by:
    - Increasing communication
    - Increasing feedback
    - Decreasing bureaucracy
    - Decreasing iteration length
  - Recognize the difficulty in separating design and construction for software
  - Recognize the extremely high rate of change in software requirements
  - Strive for asymptotic cost of late changes

The Agile Solution

- ➔ Requirements Instability
  - ➩ Short, feature-driven iterations of partial functionality
  - ➩ Maximized deployed business value
- ➔ Requirements Diversity
  - ➩ Make ongoing customer interaction part of the process
- ➔ Requirements Analyzability
  - ➩ De-emphasize formal requirement models
  - ➩ Prefer working understanding and concrete evaluation
- ➔ Incomplete/Omitted Requirements, Lack of Knowledge, Ambiguous Requirements
  - ➩ Bring the customer into the development team
- ➔ Emergent Dependencies, Conditional Requirements
  - ➩ Short iterations for more frequent risk analysis

The Agile Problem

Scalability

- ➔Research Question: How to increase scalability of agile software development without creating process burdens

Agility and Discipline

- Process Maturity
  - Ad Hoc -> Repeatable -> Defined -> Managed -> Optimizing
- Process advocates equate Agile with Ad Hoc
- But Agile is highly disciplined
  - Requires personal and team discipline
  - No process-enforced discipline
- Process has become ritualized, inward-looking
- Process is "process conformance" driven
  Agile is "customer satisfaction" driven
- Process is "contract" oriented
  Agile is "service" oriented

Agility and Discipline (2)

- Process is a religion
  Agile is a philosophy
- E.g., Documentation:
  - Process:
    - Everything is documented
    - Documentation is continuously maintained
    - Traceability is essential
  - Agile:
    - Treat documentation as a tool
    - Evaluate the cost of creation and maintenance
    - Discard it when it ceases to be useful
    - Build systems with clear designs
  - So agile development efforts don't use documentation, right?
  - Actually, many agile projects use documentation
    - Lightweight
    - Long-lifespan
    - Easy to maintain, if necessary to keep

Agile Software Development “Methods”

- Agile Database Techniques
  - Scott Ambler
- AM (Agile Modeling)
  - Scott Ambler
- Adaptive Software Development
  - Jim Highsmith
- Crystal
  - Alistair Cockburn
- FDD (Feature-Driven Development)
  - Jeff De Luca, Peter Coad (Contributor)
- DSDM (Dynamic Systems Development Method)
  - Industry Consortium (Oracle, British Airways, AmEx, etc.)
- Lean Software Development
  - Mary Poppendieck, Tom Poppendieck
- Scrum
  - Ken Schwaber, Jeff Sutherland, Mike Beedle (Contributor)
- TDD (Test-Driven Development)
  - Kent Beck
- Xbreed
  - Mike Beedle
- XP (Extreme Programming)
  - Kent Beck, Ward Cunningham, Ron Jeffries

Lecture Outline

- → Quick Review
- → Agile Software Development
- → Extreme Programming (XP)
- → Test-Driven Development
- → Communication and Documentation
- → Iterative and Adaptive Design
- → Agile Maintenance
- → Concepts for Intent-Aware Tools in an Agile context

Extreme Programming (XP)

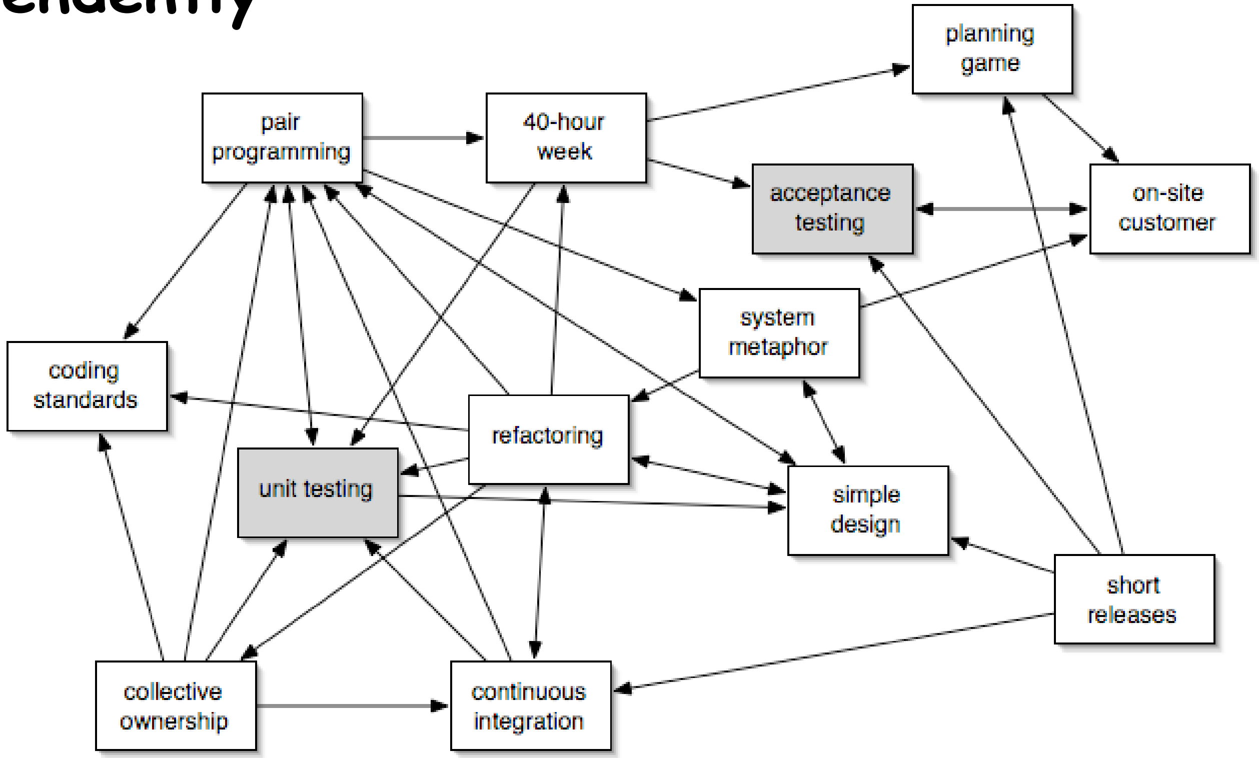
- XP is the most visible (and most viable?) form of agile software development
- Developed adaptively on the Chrysler C3 project by Kent Beck, *et. al.*
- Consists of several interacting techniques
  - Compared to Crystal, which emphasizes adaptable subsets of techniques
- Emphasizes technical and collaborative
  - Compared to Scrum, which emphasizes team management
- Core techniques updated continuously
- Most empirical research (and criticism) of agile software development features XP practices
  - Pair Programming, On-Site Customer, Refactoring

What's So Extreme?

- XP takes ideas it as far as they can go
- If testing is good, write tests first
- If code inspections are good, conduct them continuously
- If requirements documentation helps programmers understand the customer, put the customer in the middle of the process
- If prototyping is good, build working functionality on short iterations

XP Practices Overview

- → XP was designed and evolved so that the practices are interdependent
- → However, they were developed separately and offer benefits independently



XP Core Practices

- ➔Fine-Scale Feedback
  - ⇘Test-Driven Development
  - ⇘The Planning Game
  - ⇘The Whole Team
  - ⇘Pair Programming
- ➔Continuous Process
  - ⇘Continuous Integration
  - ⇘Design Improvement
  - ⇘Small Releases
- ➔Shared Understanding
  - ⇘Simple Design
  - ⇘System Metaphor
  - ⇘Collective Code Ownership
  - ⇘Coding Standard
- ➔Programmer Welfare
  - ⇘Sustainable Pace

XP Feedback Practices

- Test-Driven Development*
  - Write programmer (unit) tests an customer (acceptance) tests before planning a solution
- The Planning Game*
  - Process of selecting development priorities for an iteration
- The Whole Team
  - Bringing a customer representative into the development workspace
  - A single customer unifies requirements diversity but may be impractical
  - A team of customer specialists may not integrate with the development staff
- Pair Programming*
  - Continuous design and development feedback
  - Argumentative design space exploration

XP Process Activities

- Continuous Integration
  - The system should pass all tests and compile at all times
  - Different from “Releases”
- Design Improvement*
  - Refactor
  - Fix “bad-smelling” code
  - Eliminate unused code
- Small Releases
  - A Release comes at the end of a set of iterations
  - Deployable functionality
  - Short iterations guarantee close progress monitoring
  - Short release cycles provide additional functionality

XP Comprehensibility Practices

- Simple Design*
  - Code should be readable without comments
  - Don't build for functionality you don't know about
  - Functionality should not be repeated
- System Metaphor*
  - Instead of a solution-space architecture
  - A story of how the system works
  - Least developed, adopted core technique
- Collective Code Ownership
  - Each developer works on each part of the system
  - No/Little specialization
  - Any developer can make a change to any part of the system
- Coding Standard*
  - Formatting, Naming, Idioms, Patterns

XP 2nd Ed. and Corollary Practices

- Sit Together
  - Common, open development workspace with no partitions
- User Stories
  - Like a use-case or scenario
- Incremental Design
- Ask the Code
  - Code should be readable, intent inferable
- Spike Solution
  - Feature oriented development
  - Neither top-down or bottom-up
- Lazy Optimization and Early Profiling
- Constant Velocity
  - Make progress every day
  - No overtime death marches

Lecture Outline

- → Quick Review
- → Agile Software Development
- → Extreme Programming (XP)
- → Test-Driven Development
- → Communication and Documentation
- → Iterative and Adaptive Design
- → Agile Maintenance
- → Concepts for Intent-Aware Tools in an Agile context

Test-Driven Development

- Create test cases up-front, then write code that causes the test case to pass
- Two kinds of tests:
  - Programmer (Unit) Test
    - Specific test of a module
    - Drives the design and implementation in the solution space
  - Customer (Acceptance) Test
    - Scenario specification
    - Tests conformance to end-user (black-box) requirements
- Relies on developer and team discipline rather than process discipline

TDD Process

- Think about what and how to test
  - Yes, there is a plan!
- Write a small initial test
  - Explore the interface
  - Write enough code to compile and fail
  - Write enough code to pass (Simplest design possible)
- Write the next test
  - Develop the functional requirements
  - Write code to pass *all* tests
  - Refactor if necessary
  - Repeat
- If you want to add code to an existing module:
  - Write a test that fails under the current implementation
  - Write code to pass all the tests

Benefits of TDD

- Quality
  - Up-front testing means that QA is not an afterthought
  - Using an automated framework leads to continuous testing
- Maintenance
  - Design tests become regression tests across refactorings
- Project Management
  - Acceptance tests provide feedback on progress for iteration
- Documentation
  - A test suite is a kind of "programmer's intent" model
  - "The act of writing tests first is an act of discerning between design decisions" (Robert C. Martin)
- Design Quality
  - Incremental implementation -> incremental conceptual model
  - Testable code is decoupled, allows for ease of refactoring

TDD Idioms

- Starter Test
  - Not a realistic test, doesn't do anything
  - Where does operation belong?
  - What is its interface?
- Explanation Test
  - A test that explains how an operation should work
  - Drives design and development
- Learning Test
  - A test that explores legacy code
  - A type of integration contract
- Regression Test
  - Facilitates impact of change analysis
  - Constrain refactoring activities

TDD Patterns

- Small Tests
  - Making code testable leads to simple, comprehensible designs
- Mock Objects
  - Implement top-down, deferring design until the requirements are better understood
  - Create stubs to simulate functionality until then
- Self-Shunt
  - Allow test case to simulate external interactions
  - Tests read better
  - Explores interface design
- Logging
- Forced error conditions
- Leave the last test broken
  - It's what you were working on when you stopped for the day

Approaches to Testing

- Black Box Testing
  - Input Coverage – All combinations of legal input values
  - Expensive - exponential growth of input space
  - Excessive - congruency of groups of inputs
  - Incomplete – out of bounds/illegal inputs
- White Box Testing
  - Data-flow, control-flow coverage
  - Attempt to create minimal, adequate test suite
  - Inflexible – small code changes render test suite inadequate
  - Undecidable – due to loops and unreachable code
- Test-First Testing
  - Code and data structures are driven by the test cases that necessitate them
  - Designed for testability, reachability
  - The goal is to create code that can be tested, not tests that can exercise code
  - Test reflect design intent, not just exhaustive coverage

Testing and Specifications

- Types of specifications
  - Static
    - Compiler-enforced language constructs
  - Isolated Testing
    - Specialized execution to explore specific concepts
    - Not complete on inputs, data flow, control flow
  - Runtime checking
    - Assertions
    - Design by Contract
    - Not necessarily complete
  - Formal and Semi-formal specification with generation
    - Define a complete specification
    - Generate assertions, test cases, etc. to check correctness
    - May be white-box (control flow, data flow)
    - Or black-box (input coverage)

TDD Observations

- Test-Driven Development is an example of how Agile is *not* Ad Hoc
- Creates a kind of documentation artifact which is:
  - Useful for the life of the project
  - Automatable
  - Clearly tied to the source code (i.e., the design)
- Choice of test cases impacts design
  - But no research yet on how test selection impacts iterative design qualities
- Test cases provide a rudimentary intent and change rationale model
  - But still desiccated like source code
- TDD is non-methodical and tests are difficult to check for consistency, completeness, etc.

The Plan for Next Time

- Organizational memory and the social patterns of XP
- The cost of producing, maintaining documentation
- The dramatic tension of agile design
  - YAGNI vs. DOGBITE
- Guiding refactoring through design rationale
- Programmatic approaches to specifying design intent
- Maintainance of test suites
- Ideas for a prototype implementation of an intent-aware, agile-friendly IDE