Design Intent in an Agile Context
Part II

Paul S Grisham
grisham@mail.utexas.edu

Feb. 21, 2006

© 2006, Paul S Grisham

Review (Agile SD)

- Agile Software Development attempts to manage changing requirements through:
  - Increasing communication
    - Close, face-to-face collaborations
  - Increasing feedback
    - Short iterations with measurable progress
  - Decreasing bureaucracy
    - Self-organizing teams
  - Increasing customer satisfaction
    - Quality through innovative craftsmanship
- But what about project scalability?

Review (XP)

- Extreme Programming (XP) is an approach to agility
  - Test-Driven Development
    - Write test cases before design or code
    - Continual regression testing
  - Design Improvement
    - Refactor to improve design
  - Simple Design
    - Do the simplest thing that would possibly work
  - System Metaphor
    - Create a shared story of how the system works
- Core techniques are technical and collaborative
  - Not process- or management-oriented
- Techniques provide isolated benefit if adopted independently
  - Research on synergistic benefits weak

© 2006, Paul S Grisham

Review (TDD)

- Create test cases up-front, then write code that causes the test case to pass
  - Programmer (Unit) Test
    - Drives the design and implementation in the solution space
  - Customer (Acceptance) Test
    - Tests conformance to end-user (black-box) requirements
- 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
- Test-Driven Development is:
  - non-methodical
  - difficult to check for consistency, completeness, etc.

© 2006, Paul S Grisham

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-Driven Development in an Agile context

© 2006, Paul S Grisham

Knowledge and Communication

- Organizational Memory:
  - The sum of the knowledge of members of the organization
  - Primary location is the individual, but also
    - Textual information (manuals, databases, etc.)
    - Culture, Process, Structure
- The success of a project depends on
  - Knowing what you know
    - Does someone within the organization have the information?
  - Knowing where to find it
    - Who within the organization has the information?
- *Communication* is the means by which knowledge in the organization is transmitted to those who need it
- *Documentation* is the process of preserving knowledge in an external record

© 2006, Paul S Grisham

Communication in a Project

- Communication provides the basis of team organization
  - Team size is determined by communication complexity
  - XP Team: 1 customer, 1 manager, up to 10 developers
  - Brooks's Surgical Team: 10 members
  - Communication Complexity

$$\binom{12}{2} = 66$$

- A small group of like-minded individuals can share a mental model of the organizational memory
  - e.g., UNIX development team
- A larger project requires hierarchical organization
  - How do we coordinate communication between 100 people?

Other Problems with Individual Memory

- Knowledge Loss
  - The sole holder of some knowledge leaves the organization
- Cognitive Dissonance
  - Disagreements or conflicts between independent knowledge
- Persistence of Memory
  - Knowledge fades or changes over time
- Knowledge Dissemination
  - A holder of knowledge must respond promptly to requests
- Knowledge Organization
  - A seeker of knowledge must know where to find it
- Knowledge Maintenance
  - Changes to facts must be distributed to all who need it
- Knowledge Abstraction
  - Complex knowledge is reduced for comprehensibility

© 2006, Paul S Grisham

Documentation

- By creating an external representation for organizational memory:
  - Knowledge is preserved
  - A copy of knowledge can be distributed to each member
  - Knowledge is structured to make finding and querying easy
- Problems:
  - Cost of adding knowledge increases over time
    - Consistency checking
    - Removing invalidated facts
  - Comprehensibility is not positively related to length
    - More is not better
    - Proper scope must be maintained
    - Re-organization may be necessary
  - Very hard to get it right the first time
    - Documentation is a software project in its own right

© 2006, Paul S Grisham

The Traditional Approach

- Make documentation a deliverable
  - Documentation is a separate side-project
- Have dedicated documentation specialists
  - Harlan Mills's Surgical Team has 2 documentarians (20%)
    - Compared to 30% development team and 10% test team
- Documentation provides an interface between phases
  - Requirements -> Documentation -> Design
  - Design -> Documentation -> Implementation
- Documentation provides an interface between teams
  - API or Module Interface descriptions
  - Design intent
- Documentation preserves history
  - Design rationale
  - Change logs

Mountains of Paper!

- Brooks:
  - OS/360 project
  - Six months into the project:
    - Project workbook was five feet thick
    - Daily change updates were 2 inches thick (150 pages)
    - Workbook maintenance costs were significant
- New solution: Electronic distribution
  - Change the PDF on the server and update the whole project
  - Problem: who reads all this stuff?
    - Doesn't address the problem of getting the right information to the right people at the right time
  - Addresses accidental complexity of paper distribution
  - Does nothing to address cost of consistency maintenance

© 2006, Paul S Grisham

The Agile Approach

- Prefer working software to comprehensive documentation
- XP says very little about documentation
  - Omission does not imply elimination
  - Developers responsible for maintaining documentation
- In practice:
  - Prefer communication and shared memory to explicit docs
    - Pair programming
    - Customer collaboration
  - Code should be readable for design and intent
    - Ask the Code!
    - No code ownership means the whole team sees all the code
  - Plans and designs are intended to be temporary, then discarded
    - Cheap to produce
    - No maintenance costs

Cognitive Impact of XP's Organization

- Customer and Developer share mental model
  - Traditional development:
    - Doc. provides an interface between customer and developer
    - Customer unburdened by solution space
    - Developer receives filtered version of problem description
  - XP development:
    - Customer and Developer share mental picture of solution and problem space
    - Customer can appreciate solution challenges and costs
    - Face-to-face conversations are more efficient in transferring information
- Planning happens interactively
- Design happens iteratively (and sometimes implicitly)
- Intent modeling is an additive process of recording design choices and compromises

Process Artifacts

- The costs of process artifacts:
  - The cost of producing them initially
  - The cost of keeping them up to date
  - The cost of not keeping them up to date
- Benefits of process artifacts:
  - Support planning activities
  - Reduce detail complexity
  - Aiding comprehension (functional and design intent)

Example: Source Comments

+ Easy to produce
+ Annotate source code
+ Capture many kinds of information

- Can be hard to interpret
- No correctness, consistency checking
- No scoping information
- Incorrect, out of date comments can be dangerous

```
void Resource::readTableCompResource() {
if (_resourceFile->readUint32BE() != 'QTBL')
   error("Invalid table header");

_resourceFile->read(_versionString, 6);
_resourceFile->readByte(); // obsolete
_resourceFile->readByte(); // obsolete
_compression = _resourceFile->readByte();

readTableEntries(_resourceFile);

}
```

```
static int compareBobDrawOrder
     (const void *a, const void *b)

{
const BobSlot *bob1 =
   *(const BobSlot * const *)a;
const BobSlot *bob2 =
   *(const BobSlot * const *)b;

int d = bob1->y - bob2->y;

// As the qsort() function may
// reorder "equal" elements,
// we use the bob slot number
// when needed. This is required
// during the introduction, to
// hide a crate behind the clock.

if (d == 0) {
   d = bob1 - bob2;
}
return d;

}
```

© 2006, Paul S Grisham

Artifacts in Project Org. Memory

- Source Code
  - *Executable Specification*
  - System can be automatically generated from sources
    - Sources + Compiler + Execution Platform = System
  - Changes to sources -> Changes in system
  - Not expressive enough to express intent or rationale
    - "Dessicated"
- Tests
  - Constrain source code
  - Can be automatically executed to determine conformance
  - If a test no longer represents the intent of the system:
    - The test may fail
      - ✓ Either fix source code or repair or remove test
    - The test may not fail
      - ✓ Cost of executing an unnecessary test repeatedly

Other Artifacts

- Version Management System and Change Logs
  - Record of Design History
    - WHAT changes stored as source code delta
    - WHY recorded as natural language comments
  - Difficult to reconstruct context of changes
    - Are multiple check-ins related?
    - What is the scenario affected by the changes?
    - Use as an impact of change analysis tool is limited
- Naming Conventions
  - More likely to be used to determine intent than code itself
  - A bad name can reduce program comprehension

User-Story Cards

Customer Story and Task Card BIW Development \ COLA

DATE: 3/19/98 TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST

STORY NUMBER: 1275 PRIORITY: USER: ___ TECH: ___

PRIOR REFERENCE: ___

RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION:
SPLIT COLA: When the COLA rate chgs in the middle of the BIW Pay Period we will want to pay the 1ST week of the pay period at the OLD COLA rate and the 2ND week of the Pay Period at the NEW COLA rate. Should occur automatically based on system design.

NOTES:
For the OT, we will run a m/frame program that will pay or calc the COLA on the 2ND week of OT. The plant currently retransmits the hours data for the 2ND week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in DE Ent Excess COLA BIN.

Date	Status	To Do	Comments

Task Cards

Engineering Task Card

Smalltalk / Future

BIW Based on Conversation w/ REB: AMA **NEW**

DATE: 3/17/98

STORY NUMBER: X923 SOFTWARE ENGINEER: ____ TASK ESTIMATE: ____

TASK DESCRIPTION:
Composite Bin - Regular Base Needs to Be Displayed on GUI. We have the hidden bin for Regular Base (Lost Time) to display NOT the autogen bin but the BIN that composites the Auto Pay: the Lost Time. There is a separate composite bin started that needs to be completed??

SOFTWARE ENGINEER'S NOTES:

TASK TRACKING:

Date	Done	To Do	Comments

Using Iteration Context for Intent

- Idea: Use iteration story + test cases as design intent and rationale
  - A user story represents a cross-cutting description of a functionality
  - The development environment should be aware of the iteration context (user story)
  - Within an iteration, new tests are user-story bound
  - New tests fail; new code satisfies the test
  - Change rationale captured as the set of previously failing tests satisfied by the new code
  - Profiler determines scope for feature aspects
  - Refactoring micro-iterations identified explicitly by user input or as iterations with no new tests satisfied

Using Test Cases as Intent Model

- Any source element can be queried with respect to:
  - User stories it participates in
  - Test cases it participates in
  - Classes/Methods that call it
  - Classes/Method it calls
- Test cases can be queried with respect to:
  - Various levels of code and data flow coverage
  - User stories it participates in
  - Former test cases it supercedes
  - Stubbed portions of the implementation (completeness)

Characteristics of Agile Documentation

- Cost of initial production must be low
  - Should be able to be done by developers
  - Non-interfering
- Prefer monotonic representations of knowledge
  - Low cost/risk of dissonance
  - New information replaces old automatically
- Prefer documentation that drives or is derived from the executable specification
  - Minimal amount of manual traceability management
  - Single underlying representation for active artifacts

© 2006, Paul S Grisham

Characteristics of Agile Documentation

- Lifespan should be definable
  - (Utility over time function)
    - Information is relatively static (information long-lasting),
    - Cost of updating is cheap (information continually useful), or
    - Planned retirement (limited lifespan)
- Think about who will use it and how
  - No documentation for documentation's sake
  - Developers know what they need
  - Similar to agile design, if you don't know how something will be used, don't bother

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-Driven Development in an Agile context

© 2006, Paul S Grisham

What Is Design?

- Design Process
  - Requirements are partitioned into elements
  - Design specifies relationships on those elements
  - Requirements are be decomposed into sub-requirements
  - Implementation is the process of refining reqs. and design
- Design can take place at various levels
  - Instruction, Statement, Function, Object, Component, etc.
- Hierarchical designs are easier to comprehend
  - An architectural diagram can make sense of 100+ classes
- Detailed designs drive cost estimation
  - Predictive model?
- Traditional design:
  - Top-down design
  - Bottom-up implementation
  - Tight internal cohesion
  - Loose external coupling

Big Up-Front Design (BUFD)

- ➔ If requirements are static
  - ↳ Designs can be definitive
  - ↳ Can optimize the design for desired qualities
  - ↳ Complex inter-dependencies are tolerable
  - ↳ No risk of spending up-front time building bottom-up
- ➔ If requirements are volatile
  - ↳ Designs will have to evolve to accommodate changes
  - ↳ Evolved designs may not preserve original design qualities
- ➔ If requirements are poorly understood
  - ↳ Bottom-up development may not help clarify requirements
  - ↳ Requirements faults not found until late

© 2006, Paul S Grisham

Agile Design: Simple and Iterative

- In XP, a given user story is selected for development
  - A story must fit in an iteration or be subdivided into new stories
  - Stories are decomposed into Engineering Tasks
  - Acceptance Tests are written (customer-owned)
- Using TDD and Simple Design, a solution is created until all acceptance tests pass
- The customer gets rapid feedback on the implementation, and can make corrections to requirements
- When design elements need to be merged, conduct a refactoring full iteration to improve design
  - Refactored design should reflect real need
  - Use regression tests to maintain correctness

© 2006, Paul S Grisham

XP Design Planning

- XP Projects use design planning
  - e.g., Between teams within an iteration
  - Try a pilot implementation before refactoring
  - During refactoring, if you can see the need, implement flexibility for future enhancement
    - Don't invent new tasks!
    - Would it be better to increase the priority of that story?
- YAGNI vs. DOGBITE
  - Ya Ain't Gonna Need It
    - Might be nice to have, but odds are YAGNI
    - Wait until you have a user story to support it
      - ✓ Complex character encoding support for a small business system
      - ✓ User-customizable reports
      - ✓ DBMS brand independence
  - Do it Or it's Gonna Bite you In The End
    - You have a real fork in the road between incompatible options
    - You need to attend to some pervasive quality requirement
      - ✓ Security, Multi-Threaded
      - ✓ Scalability (?)

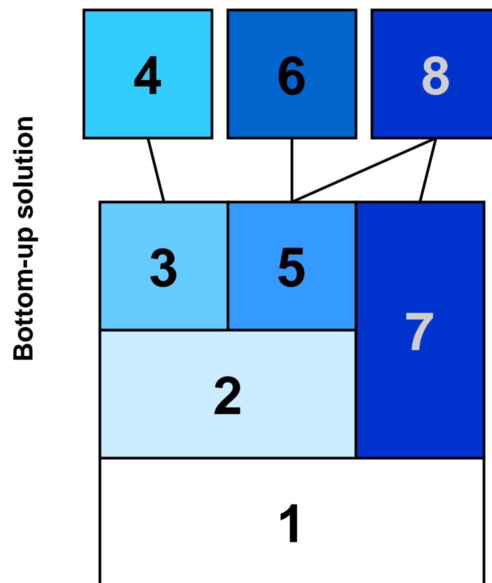
Example

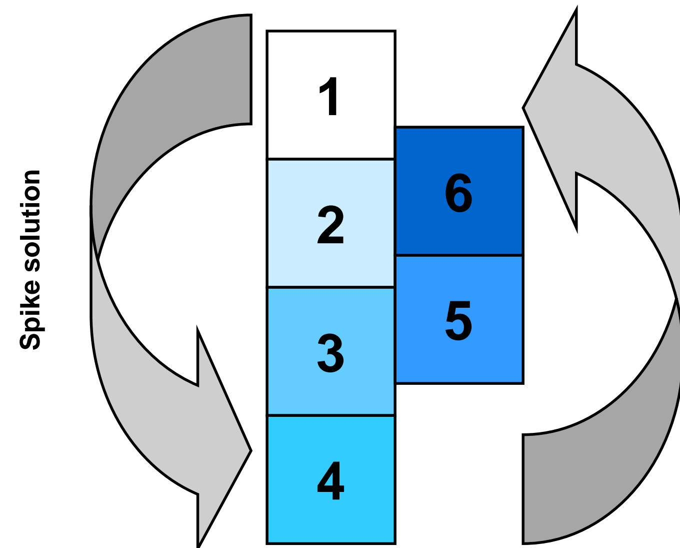
- First iteration:
  - Some data must be stored to disk
  - Simplest solution is a text file
- Second iteration:
  - Simplest solution is to store to disk in another file
  - However code already exists to write a file to disk
  - Refactor to reuse the file interface
- Third iteration:
  - Requires coordination between data in 1st and 2nd iteration
  - Marked-up data seems reasonable -> XML
- Fourth iteration:
  - We need multi-user and transaction support
  - Consider a relational DBMS
  - Now we know what are data requirements are
  - We can write scripts to import our XML

© 2006, Paul S Grisham

Spike Solution

- Sometimes we need to try a solution without knowing what the requirements are
- Spike Solution is an end-to-end experimental solution
  - Depth-first, top-down
  - Independent of existing solutions
  - Close the Loop





XP and Prototyping

- Prototyping – an experimental implementation designed to get rapid feedback
- Although XP uses small iterations and incremental development of functionality, it differs from prototyping in critical ways:
  - Designed to be functional
    - No mockups
  - Code is production quality
  - New features are integrated into the system immediately
  - Even spike solutions are meant to be refactored into the system eventually
  - Tests drive development and serve as quality and regression control

Deferred Decisions

- Sometimes the best solution isn't known or the requirements are uncertain
- Use a Design Shield to defer decisions
  - May be implemented as a façade or abstract interface
  - Change is anticipated behind the shield
  - Refactoring firewall
  - "It gives you room to change your mind"
  - The more protective it is, the more complex the design becomes
- Use placeholders or stubs to make explicit where future code will go
  - Sometimes mock functions or data will make a unit compile and pass a test

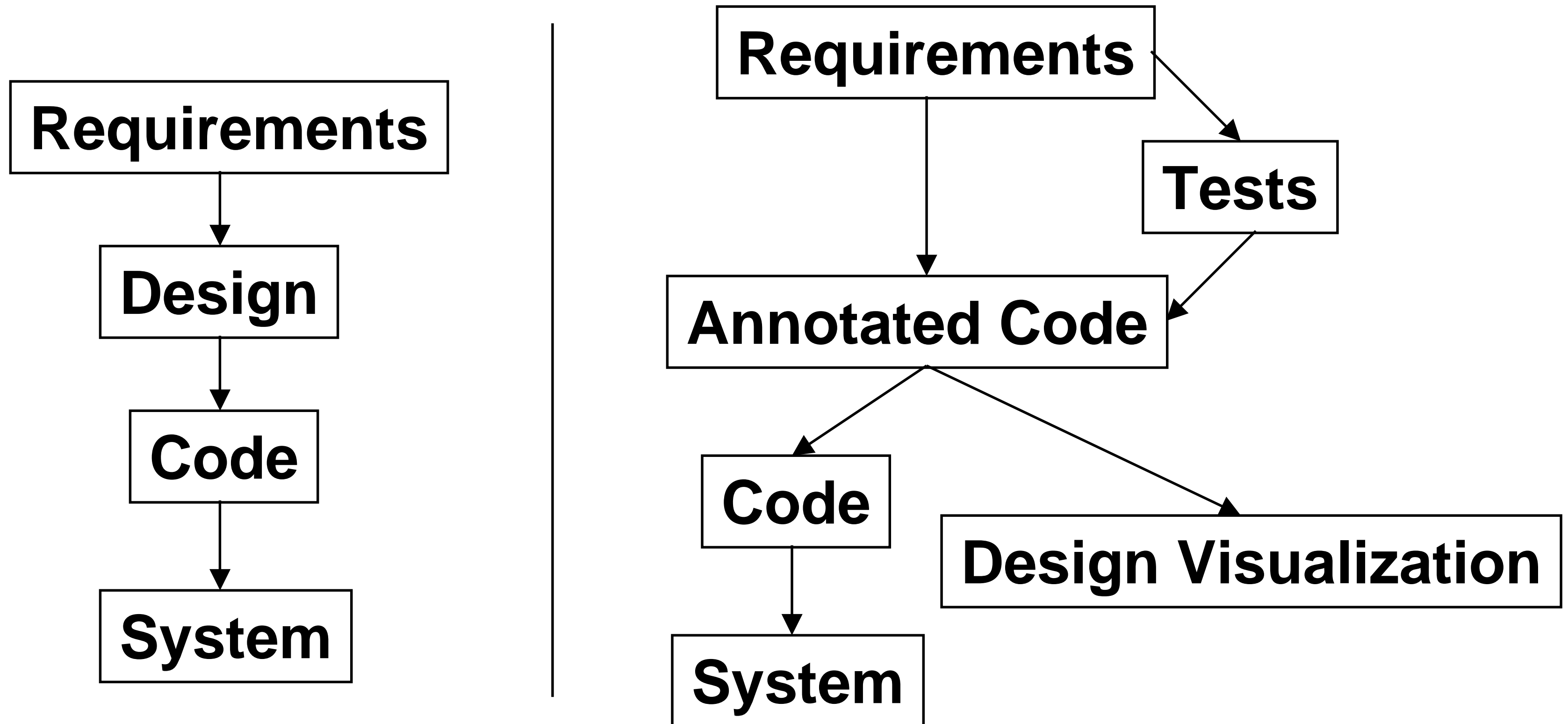
© 2006, Paul S Grisham

Role of Rationale and Intent

- Design Rationale can mark where design alternatives exist
  - Decision might have been deferred or
  - The best solution was temporarily rejected until later
  - Deferred decisions can be queried to identify
- Design Rationale bounds refactoring
  - Two or more qualities are driving refactoring -> Code churn
  - Refactor -> Refactor back -> Refactor, etc.
  - Can more easily identify the competing strategies
  - May facilitate arbitration and conclusion
- A stub is an expression of design intent
  - Marks the location where new code can plug in
  - Doesn't break the existing code
  - It's in the code, so changes to code change the design

© 2006, Paul S Grisham

Design as a Quality



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-Driven Development in an Agile context**

© 2006, Paul S Grisham

Maintenance of Test Cases

- Maintenance of Test Cases is very hard
  - Perhaps harder than code maintenance
  - Relationships between Test Cases are unclear
    - Designed to be independent, isolated
- Most work on test maintenance is on test suite reduction
  - Tests take a long time to run
  - Searching for the smallest set of tests to get "adequate" coverage
  - The basic test is a call to a code unit on a variety of inputs and compare outputs
    - Treat state and side effects as inputs
  - Problem: Small changes in the code can make test set "inadequate" -> need new test set

© 2006, Paul S Grisham

Test Case Generation

- Alternate approach: generate tests from specifications
  - For a given definition of coverage, generate "adequate" test
  - If code changes, new inputs can be generated automatically
  - Old tests can be stored for regression
  - Writing an oracle can be very hard
- Neither approach addresses:
  - Adequately testing error or out-of-bounds conditions
  - How to work with partial knowledge of specifications
  - How to update tests when requirements change
  - How to test interaction effects, such as with use cases
- These approaches to test selection provide no contextual information
  - We want to treat a test case as a model of intent
  - Path coverage may test interactions outside of design intent

Shortcomings in TDD

- Non-methodical
  - How do we know when we have enough tests?
- Difficult to maintain
  - Simple changes to code design can break lots of useful tests
- Test semantics are informal
  - Rely on idiomatic usages to express intent
  - Hard to analyze what the test is doing
    - Systematically create new tests
    - Generalize into super-tests
- Relationship between tests and other contracts is unclear
  - Assertions
  - Contracts (e.g., pre- and post-conditions)
- Selection and prioritization of test cases influences design

© 2006, Paul S Grisham

Possible Approaches

- Use a lightweight specification model to guide test creation
  - Makes test creation methodical
  - Changes to requirements can propagate easily to tests
  - Requires additional spec language
- Use a specialized test representation and framework to generalize specifications from tests
- Create explicit intent bindings from test points to unit parameters
- Bind code assets to tests (reverse traceability) in order to facilitate propagation of changes to tests

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-Driven Development in an Agile context

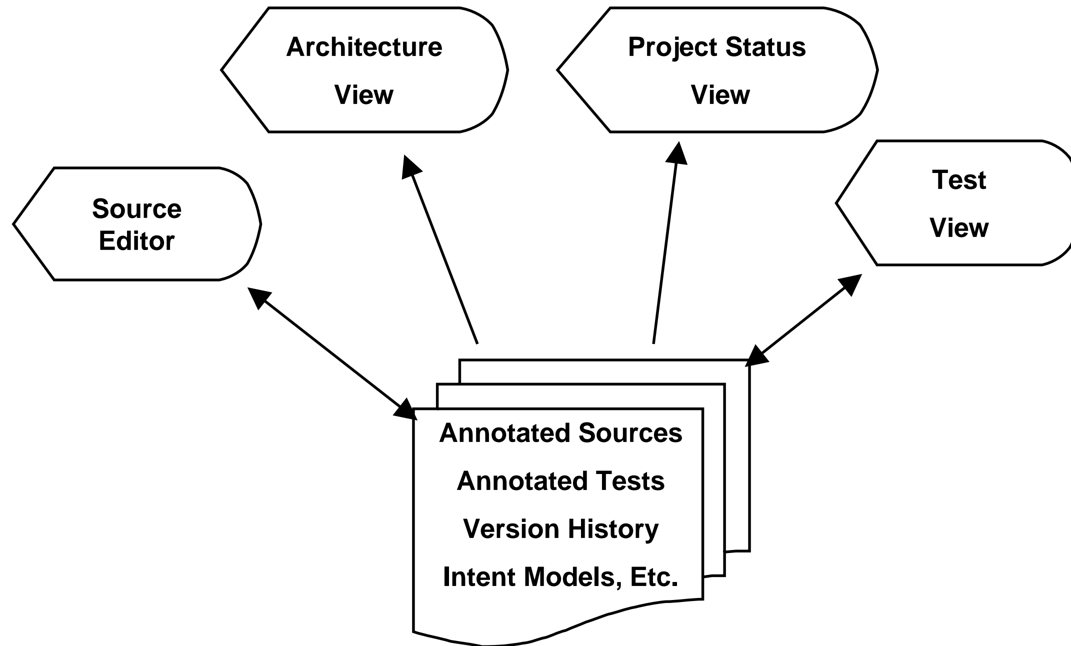
© 2006, Paul S Grisham

IDE Design Philosophy

- Total integration through views
  - Source editing
  - Version management
  - Test automation and reporting
  - Semantic interconnection visualization
  - Intent and rationale modeling
  - Design visualization
  - Progress status visualization
  - Planning
- Modeling and documentation should be as unobtrusive as possible
- Use context where possible
- Use programmer's apprentice when necessary

© 2006, Paul S Grisham

Vision of Agile Artifact Repository



- Central, managed database of system assets
- All traceability managed by development environment through semantic intent models
  - Changes easily propagated
- Some traditional planning views (e.g. architecture) are generated views rather than design drivers

© 2006, Paul S Grisham

Intent-Aware Development Environments

- Inscape
- SEURAT
  - Integrated approach to capturing design rationale
- Evolutionary Annotation Prototype
  - Use change log information to assist program comprehension
- Intentional Programming

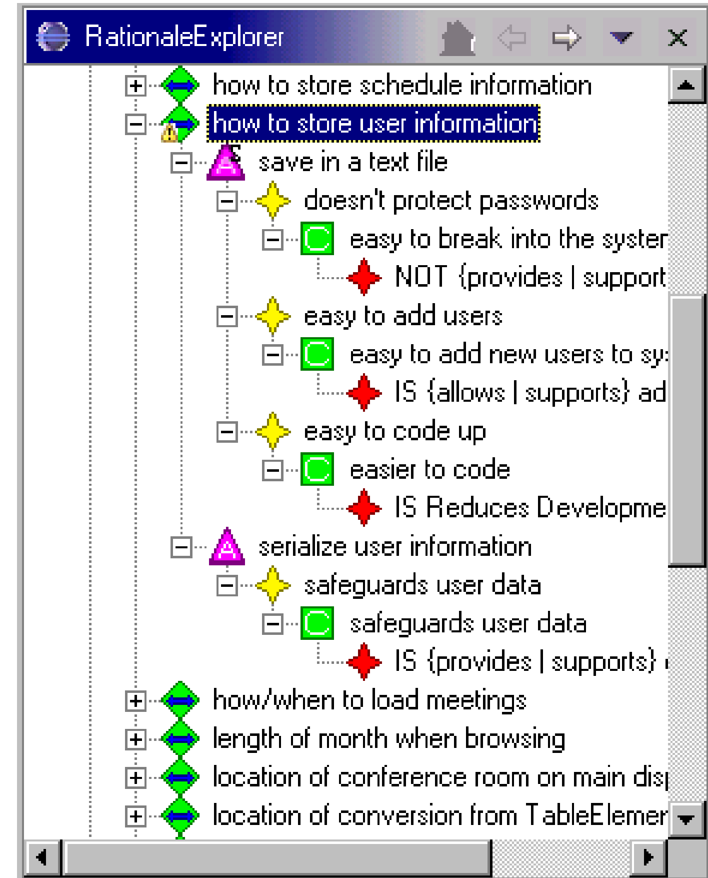
© 2006, Paul S Grisham

Inscape

- Lightweight semantic interconnection model with specification language
  - Pre-conditions
  - Post-conditions
  - Obligations
- Built on old technology (Gandalf)
- Due for an update in the color, window, language-aware editor world
- How do programmers respond to Inscape's design process?

SEURAT

- Software Engineering Using RATionale
- Ontology of design rationale behind an IDE
  - Supports management of alternative design choices
- Designed to support software maintenance
  - Not necessarily initial design
- Binds requirements to code elements through rationale
- Integrated into Eclipse



SEURAT Tool

- Allows input of information about:
  - Decisions
  - Alternatives
  - Evaluation Criteria

Decision Information

Name: how to store user information

Description: How do we store user information - names, passwords, etc.

Type: SingleChoice Status: Unresolved

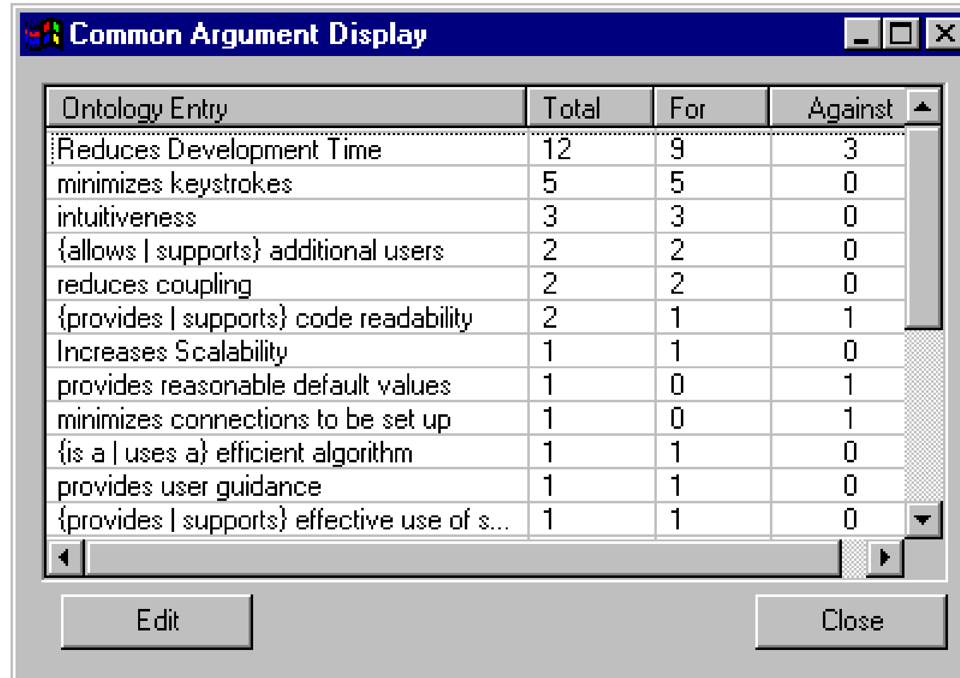
DevelopmentPhase: Design Sub-Decisions Required

(Evaluation) Alternatives:

(-6.5) save in a text file
(10.0) serialize user information

Save Cancel

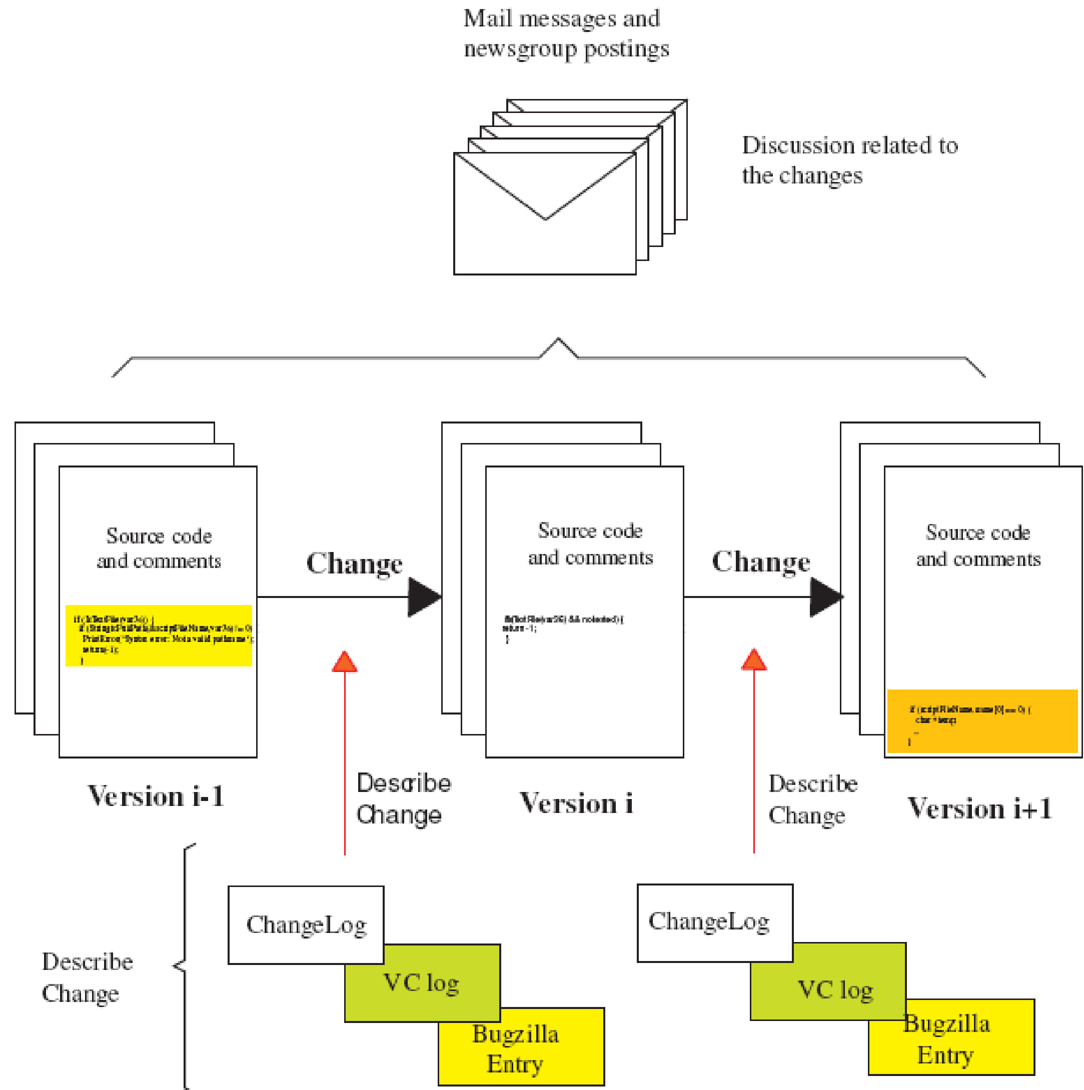
SEURAT Tool



➔ Tool can help evaluate alternatives based on the arguments for and against a particular design choice

Evolutionary Annotations

- *Submitted for publication MSR2006*
- Use change logs and version management comments to annotate source code views
- An attempt to maximize use of unstructured, natural data



EA Prototype Tool

C/C++ Projects | Navigator

- ap
- helpers
- include
- lib
- main
  - Makefile.tmpl 106443 11
  - NWGNUmakefile 106443
  - NWGNUmakefile.mak 329
  - alloc.c 106443 11/24/04
  - buff.c 106443 11/24/04
  - gen_test_char.c 106443
  - gen_test_char.dep 10644
  - gen_test_char.dsp 10644
  - gen_test_char.mak 10644
  - gen_uri_delims.c 106443
  - gen_uri_delims.dep 1064
  - gen_uri_delims.dsp 1064
  - gen_uri_delims.mak 1064
  - http_config.c 106443 11/
  - http_core.c 105350 9/30
  - http_log.c 106443 11/24
  - http_main.c 160164 4/5/
  - http_protocol.c 230826 8
  - http_request.c 88479 3/
  - http_vhost.c 106443 11/
  - rfc1413.c 106443 11/24
  - util.c 356278 12/12/05
  - util_date.c 106443 11/2
  - util_md5.c 106443 11/2
  - util_script.c 106443 11/2
  - util_uri.c 106443 11/24/
- modules
- os
- regex
- support
- test

buff.c | Makefile.tmpl | Compare http_core.c <workspace> and versions

Structure Compare

Text Compare

Workspace file: http_core.c

```
 * But then again - you should use AuthDigestRealmSeed in your c
 * file if you care. So the adhoc value should do.
 */
return ap_psprintf(r->pool,"%pI%pp%pp%pp%pp",
        &r->connection->local_addr.sin_addr,
        (void *)ap_user_name,
        (void *)ap_listeners,
        (void *)ap_server_argv0,
        (void *)ap_pid_fname);
}

API_EXPORT(const char *) ap_default_type(request_rec *r)
{
```

Repository file: http_core.c

```
 * But then again - you should use AuthDigestRealmSeed in yo
 * file if you care. So the adhoc value should do.
 */
return ap_psprintf(r->pool,"%pp%pp%pp%pp%pp",
        (void *)&((r->connection->local_addr).sin_addr ),
        (void *)ap_user_name,
        (void *)ap_listeners,
        (void *)ap_server_argv0,
        (void *)ap_pid_fname);
}

API_EXPORT(const char *) ap_default_type(request_rec *r)
{
```

Problems | Console | Properties | Progress | SVN Resource History | Search | Evolutionary Annotations

Type	Scope	TimeStamp	Author	Version	Other
email	global	2004-11-21 15:	jorton	103824	still some reg exp problems
svn commit log	global	2004-11-20 03:	nd	103824	Fix a bunch of cases where the return code of the regex (
bug 28218	local	2004-11-17 04:	Kevin J Walters <l	103824	errors in regular expressions for LocationMatch cause sile
email (code revie	global	2004-11-18 10:	Jeff Trawick, Joe (	103824	Votes +1, +1

```
The problem turned out to be the ||. I had a quick glance through the code and
it reads like the regular expression library doesn't like this and regards it
as an error (i note solaris egrep errors, perl thinks its ok). The problem is
that this error is not reported to the user so the configuration appears to be
ok when the process is started. I think this is both confused to the naive
configuration creator and potentially dangerous if the Location block contains
some critical (say, security-related) directives.

It looks like the (handful of) ap_pregcomp calls in http_core.c do not check
for a NULL return code that would indicate a failed compilation. So this
affects Location ~, LocationMatch, Directory ~, DirectoryMatch, Files ~,
FilesMatch.

Perhaps this problem exists in apache 2.0 as well? And maybe other areas of
apache 1.3 (not mod_alias, just had a look there!).
```

EA Comments

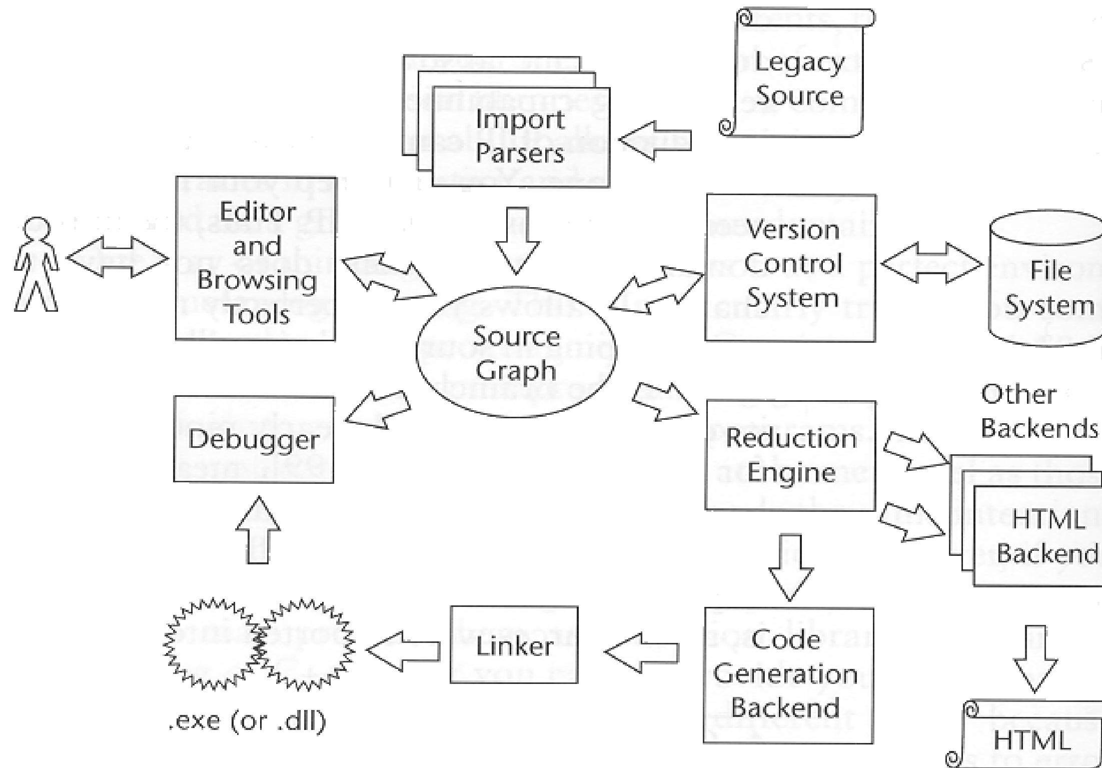
- Similar to our approach of using user-story context and test cases
- This approach relies on:
  - The availability of comprehensive version control comments
  - The ability to associate project communication to changes after the fact
- Advantages:
  - Uses existing process artifacts
  - Unobtrusive
- Disadvantages:
  - Informal model (vs. semi-formal)
  - Navigable through source organization, not usage patterns
  - No specific notion of design intent

Intentional Programming

- Developed by Charles Simonyi (formerly of Microsoft)
- Treat source code as an tree of *Active Source* elements
- Active sources may use:
  - Traditional programming languages
  - Domain specific abstractions
  - Graphical notations
- Language extensions provide:
  - Rendering Methods
  - Input Methods
  - Reduction Methods
    - Convert one format to another
  - Debugging Methods
  - Editing and Refactoring Methods
  - Version Control Methods
    - For resolving conflicts

IP System Design

- Treat Code as a Graph, not as files
- Use parsers and reduction to treat functionality in a language-independent manner



IP Screenshots

Bessel

exeonly double **Bessel**(double **x**)

{

int **n**;

return $\sum_{n=0}^{20} t(n, x)$;

}

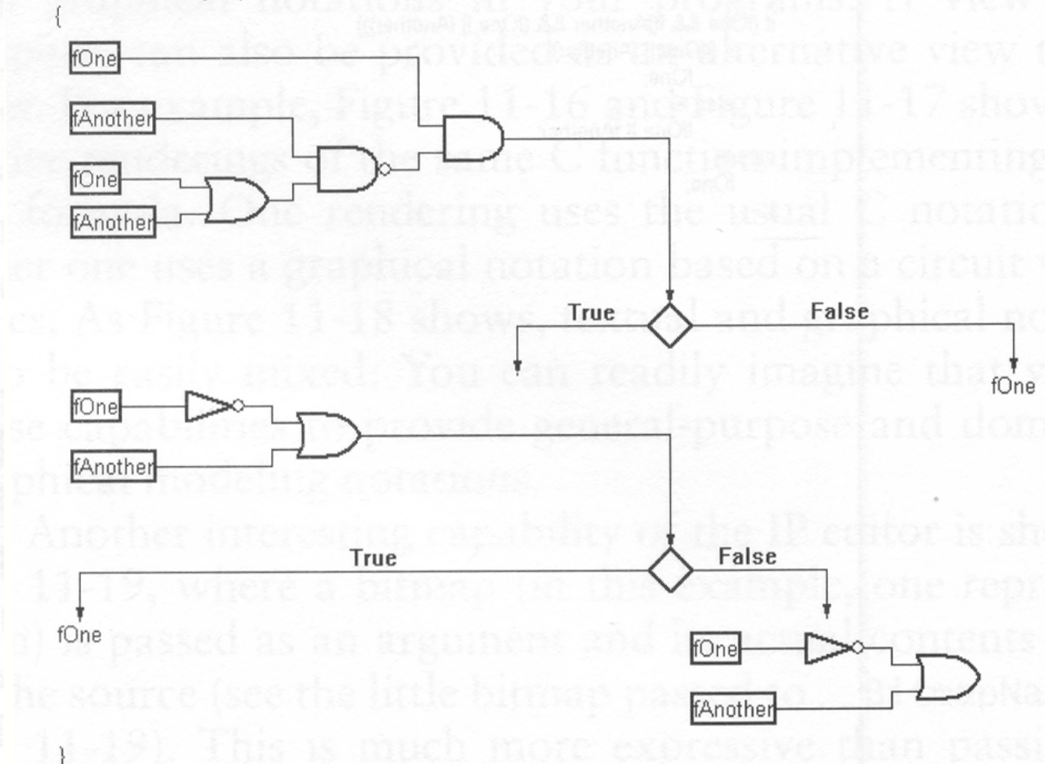
t

__private double **t**(int **n**, double **x**)

{

return $\begin{cases} 1.0 & \text{if } n == 0 \\ t(n-1, x)\left(\frac{x}{2n}\right)^2 & \text{otherwise} \end{cases}$

}



- ➔ Functional Intent very clear
- ➔ Useful for implementing domain-specific languages

Intentional Programming Comments

- Right idea on treating source code as a view into a functional intent model
- Development system is proprietary and secretive
  - Not much has been seen since Simonyi left Microsoft
- Emphasizes the importance of using the right abstractions to capture intent
- Most developers already know their favorite language
  - Otherwise we would be using better languages than C++ and Java
- Not clear how the idea generalizes to real systems

© 2006, Paul S Grisham