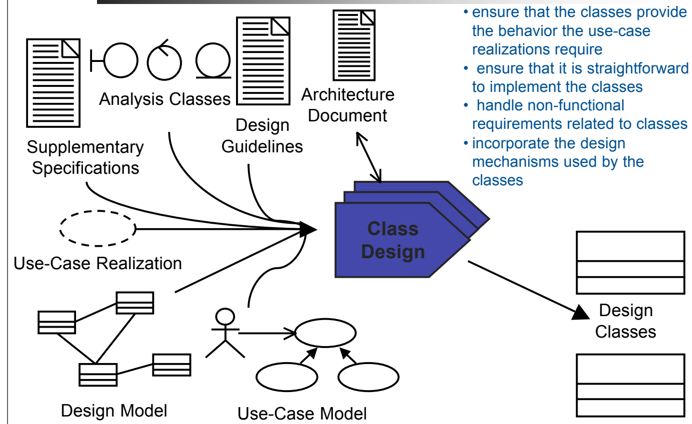
23- Design & Analysis

Rick Adrion

Class Design Overview

- ensure that the classes provide the behavior the use-case realizations require
- ensure that it is straightforward to implement the classes
- handle non-functional requirements related to classes
- incorporate the design mechanisms used by the classes



How Many Classes Are Needed?

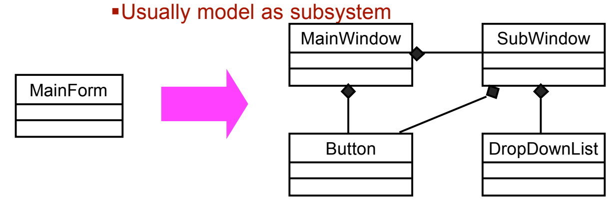
- Many, simple classes means that each class
  - encapsulates less of the overall system intelligence
  - is more reusable
  - is easier to implement
- A few, complex classes means that each class
  - encapsulates a large portion of the overall system intelligence
  - is less likely to be reusable
  - is more difficult to implement
- A class should have a single well focused purpose
  - a class should do one thing and do it well!
  - how does this relate to my earlier suggestion that classes have multiple responsibilities?

- Class should have **multiple** responsibilities
  - Actions that object can perform
  - Knowledge object maintains
  - Non-functional requirements

Designing Boundary Classes

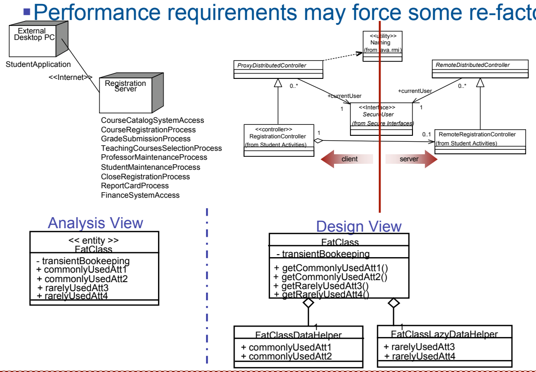
- User interface (UI) boundary classes
  - What user interface development tools will be used?
  - How much of the interface can be created by the development tool?
  - "Reverse Engineering"
- External system interface boundary classes
  - Usually model as subsystem

See Maciaszek - no time to discuss UI design



Designing Entity Classes

- Entity objects are often passive and persistent
- Performance requirements may force some re-factoring

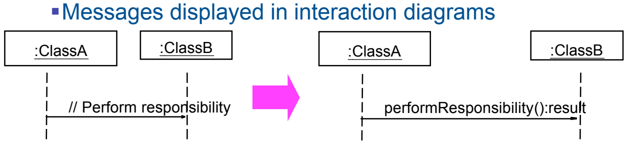


Designing Control Classes

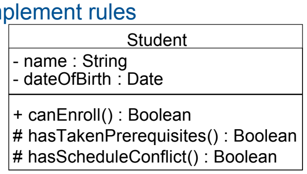
- What Happens to Control Classes?
  - Are they really needed?
    - if just “pass-throughs” from the boundary classes to the entity classes, they may be eliminated.
  - Should they be split?
    - might depend on distribution, e.g., proxy-remote
- Control classes may become true design classes for any of the following reasons:
  - they encapsulate significant control flow behavior,
  - the behavior they encapsulate is likely to change
  - the behavior must be distributed across multiple processes and/or processors
  - the behavior they encapsulate requires some transaction management.

Operations

- Messages displayed in interaction diagrams



- Implement rules

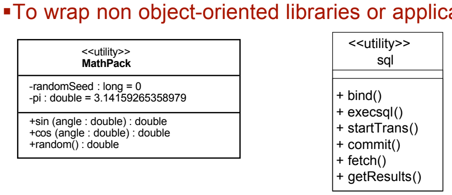


every class should have:
- *Manager functions*
- *Implementor functions*
- *Access functions*
- *Helping functions*

- Operations can lead to new class definitions

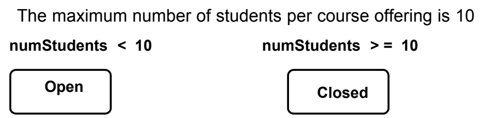
Utility Classes

- What is a Utility Class?
  - Utility is a class stereotype
  - Used for a class that contains a collection of free subprograms
- Why use it?
  - To provide services that may be (re)useful in a variety of contexts
  - To wrap non object-oriented libraries or applications

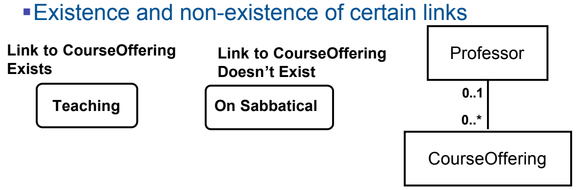


Identify and Define the States

- Significant, dynamic attributes



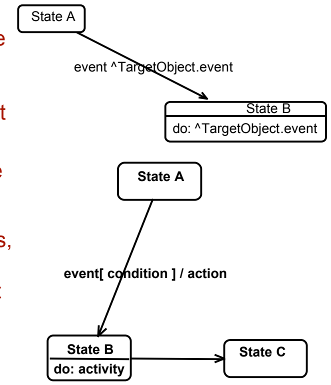
- Existence and non-existence of certain links



- explicitly define what it means to be in a particular state.

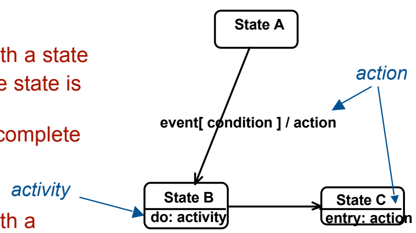
Identify the Events & Transitions

- Events
  - One event may trigger the sending of another event
  - An activity can also send an event to another object
- Transitions
  - For each state, determine what events cause transitions to what states, including guard conditions, when needed
  - Transitions describe what happens in response to the receipt of an event

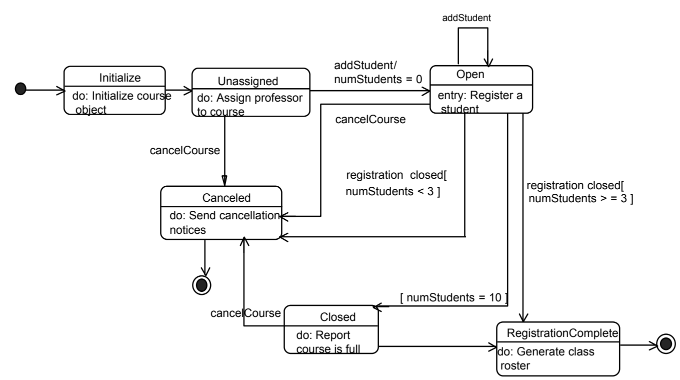


Add Activities and Actions

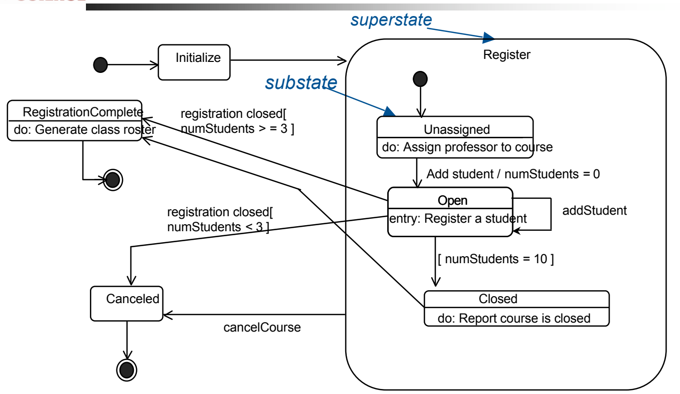
- Activities
  - Associated with a state
  - Start when the state is entered
  - Take time to complete
  - Interruptible
- Actions
  - Associated with a transition
  - Take an insignificant amount of time to complete
  - Non-interruptible



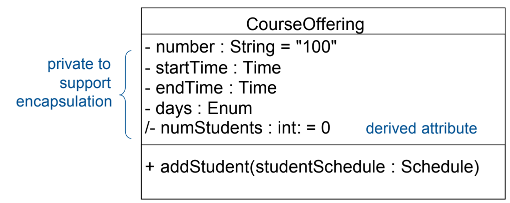
Statechart



Statechart with Nested States

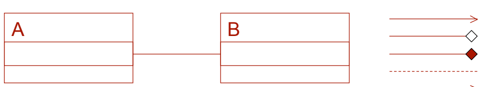


Example: Define Attributes



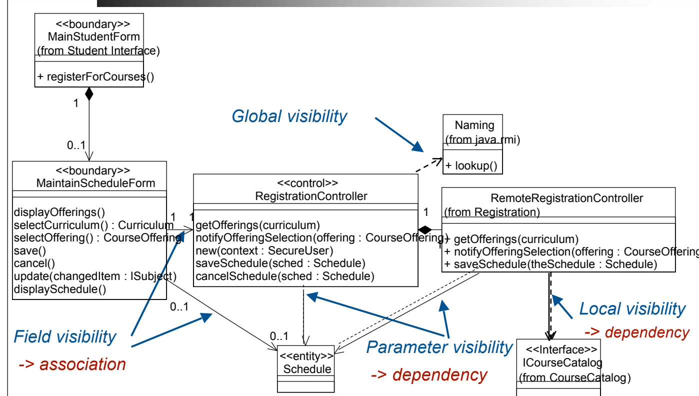
Associations & Dependencies

- association

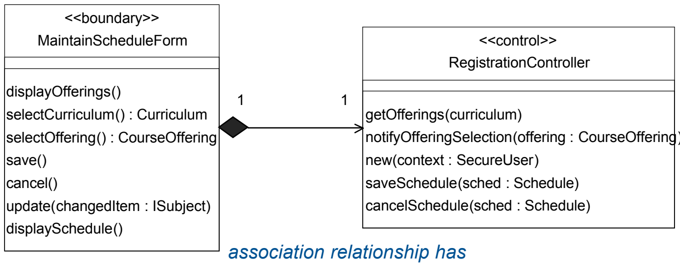


- visibility
  - attribute (field) visibility: B is an attribute of A
    - remains an association
  - parameter visibility: B is a parameter of a method A
    - becomes a dependency
  - local visibility: B is a (non-parameter) local object in a method of A
    - becomes a dependency
  - global visibility: B is in some way globally visible
    - becomes a dependency

Define Dependencies



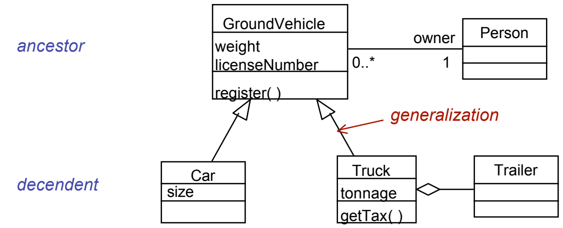
Example: Composition



association relationship has been refined into a composition relationship.
MSF represents a session
RC never exists outside of session

Generalization (Inheritance)

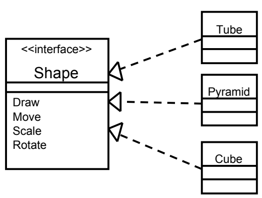
- One class inherits from another

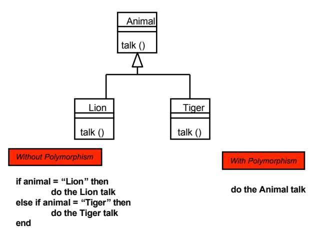


- not just finding common attribute, operations and relationships
- more about the responsibilities and essence of the classes.
- avoid "skyscrapers"; the hierarchies should look like small, independent "forests

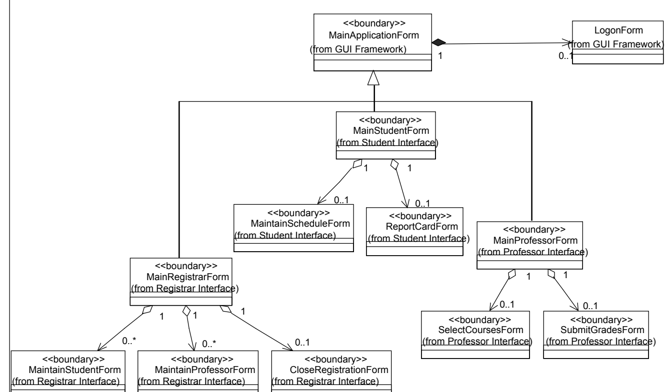
Polymorphism?

- The ability to hide many different implementations behind a single interface
- Generalization to support polymorphism



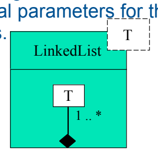


Define Generalizations

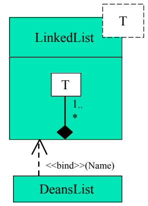


Parameterized Class

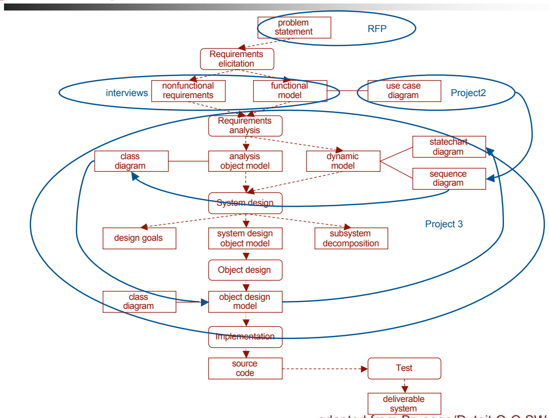
- A parameterized class or template defines a family of potential elements.
- To use it, the parameter must be bound.
- A template is rendered by a small dashed rectangle superimposed on the upper-right corner of the class rectangle. The dashed rectangle contains a list of formal parameters for the class.



- Binding is done with the <<bind>> stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.
- Here we create a linked-list of names for the Dean's List.



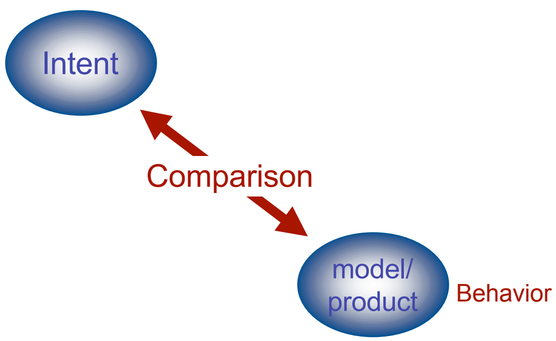
O-O System Development



adapted from Bruegge/Dutoit O-O SW Engr

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

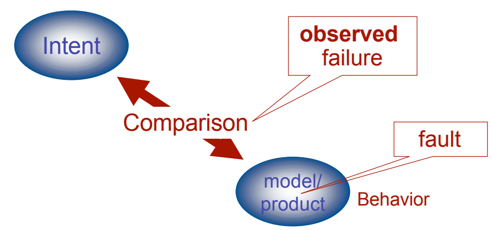
Analysis



University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Basic Definitions

- Failure-- result that deviates from the expected or specified intent
- Fault/defect-- a flaw that could cause a failure
- Error -- erroneous belief that might have led to a flaw that could result in a failure

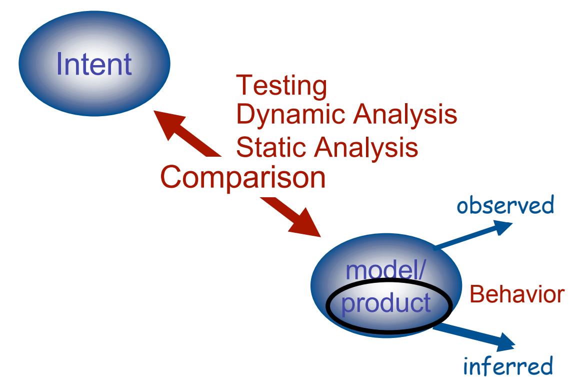


University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Approaches

- Static Analysis
  - the static examination of a product or a representation of the product for the purpose of inferring properties or characteristics
- Dynamic Analysis
  - the "interpretation" of a product or representation of a product for the purpose of inferring properties or characteristics
- Testing
  - the (systematic) selection and subsequent "execution" of sample inputs from a product's input space in order to infer information about the product's behavior.
    - usually trying to uncover failures
    - the most common form of dynamic analysis
- Debugging -- the search for the cause of a failure and subsequent repair

Analysis



Validation and Verification: V&V

- Validation -- techniques for assessing the quality of a software product
- Verification -- the use of analytic inference to (formally) prove that a product is consistent with a specification of its intent
  - the specification could be a selected property of interest or it could be a specification of all expected behaviors and qualities
    - e.g., provide a user-friendly and efficient ATM system for remotely depositing funds into and withdrawing funds from a checking or saving account
    - e.g., all deposit transactions for an individual will be completed before any withdrawal transaction will be initiated
  - a form of validation
  - usually achieved via some form of static analysis

Correctness

- a product is functionally correct if it satisfies all the functional requirement specifications
  - correctness is a mathematical property
  - requires a specification of intent
  - specifications are rarely complete
- a product is behaviorally correct if it satisfies all the specified behavioral requirements
  - difficult to prove poorly-quantified qualities such as user-friendly

COMPUTER SCIENCE

Reliability

- measures the dependability of a product
  - the **probability** that a product will perform as expected
  - sometimes stated as a property of time e.g., mean time to failure
- Reliability vs. Correctness
  - reliability is relative, while correctness is absolute
  - given a "correct" specification, a correct product is reliable, but not necessarily vice versa

COMPUTER SCIENCE

Robustness

- behaves "reasonably" even in circumstances that were not expected
  - making a system robust more then doubles development costs
  - a system that is correct may not be robust, and vice versa

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

COMPUTER SCIENCE

Formal models

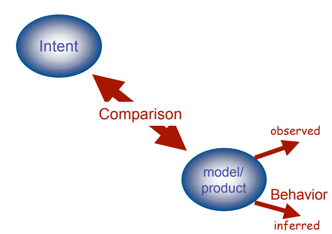
- Analysis is usually done on a model of an artifact
  - textual representation of the artifact is translated into a model that is more amenable to analysis then the original representation
  - the translation may require syntactic and semantic analysis so that the model is as accurate as possible
    - e.g., x:= y + foo.bar
  - model must be appropriate for the intended analysis
- graphs are the most common forms of models used
  - e.g., abstract syntax graphs, control flow graphs, call graphs, reachability graphs, Petri nets, program dependence graphs

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

COMPUTER SCIENCE

Modeling intent & artifacts

- natural language
- structured natural language
- pictorial notation
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- data models
- formal language(s)
  - state-oriented
  - function-oriented
  - object-oriented

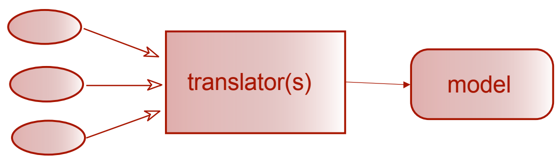


University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Ideally want general models

- different languages
  - e.g., Ada, C++, Java
- different levels of abstraction/detail
  - e.g., detailed design, arch. design
- different kinds of artifacts
  - e.g., code, designs, requirements

translate textual representations



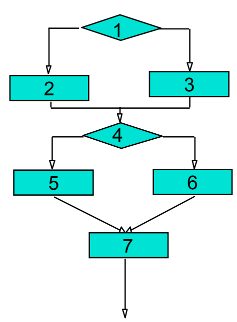
Static analysis

- typically conservative
  - never declare a property to be valid if it is not
  - usually achieve this by using representations that over-estimate actual behavior
  - the representation depends on the analysis
- AST is a conservative representation for
  - determining all the operators in a program
  - determining all the locations where X is defined
- CFG is a conservative representation for
  - Determining how many loops are in the program
  - determining how deeply nested each loop is

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

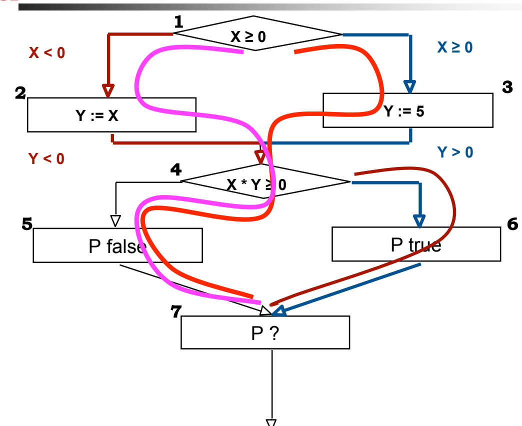
Conservative analysis in CFG

- For all execution sequences, is P true?
  - if P is true for all paths, then P is true
  - if P is true for some paths, then P may be true or false
    - Paths where P is not true may not be feasible
- For some execution sequence, is P true?
  - if P is true for some path, P may be true or false
    - the path where P is true may or may not be feasible
- Conservative analysis would only say P is true if is known to be true for all paths



University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

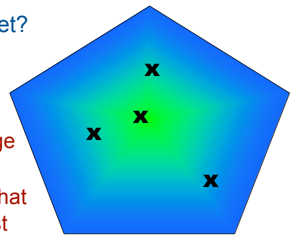
Example with an infeasible path



University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Dynamic analysis techniques

- draw inferences from a sample of the problem domain
- how do we choose that subset?
- Fault detection may depend upon
  - Specific combinations of statements, not just coverage of those statements
  - Astutely selected test data that reveals the fault, not just test data that executes the path



Approaches

- Static Analysis
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification
- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing

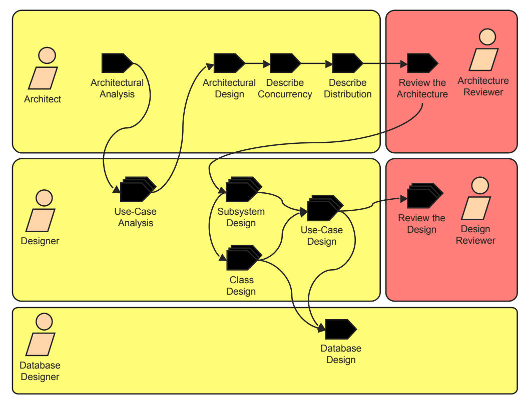
University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Reviews, Inspections, and Walkthroughs

- Manual static analysis methods
- Most can be applied at any step in the lifecycle
- Have been shown to improve reliability, but
  - often the first thing dropped when time is tight
  - labor intensive
  - often done informally, no data/history, not repeatable

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Reviews in the RUP



University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Reviews, Inspections, and Walkthroughs

- Formal reviews
  - author or one reviewer leads a presentation of the product
  - review is driven by presentation, issues raised
- Walkthroughs
  - usually informal reviews of source code
  - step-by-step, line-by-line review
- Inspections
  - list of criteria drive review
  - properties not limited to error correction
  - historical context

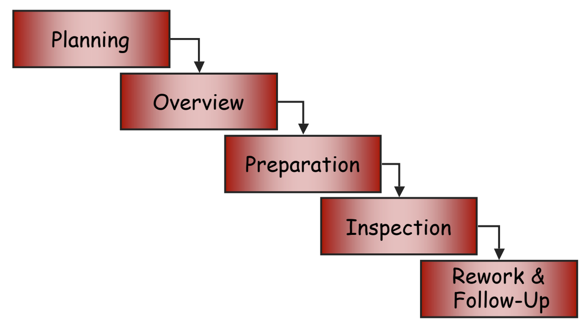
Review methods

- Fagan inspections
  - formal, multi-stage process
  - significant background & preparation
  - led by moderator
- Active design reviews
  - also called "phased inspections"
  - several brief reviews rather than one large review
  - guided by questions from the author
- Cleanroom
  - more than reviews, but reviews important component
  - we'll come back to this
- N-fold
  - parallel reviews controlled by moderator
  - focuses on user requirements

Fagan Inspections (3-5 participants)

- **Moderator**
  - **Responsible for organizing, scheduling, distributing materials, and leading the session**
- **Author**
  - **Responsible for explaining the product**
- **Scribe**
  - **Responsible for recording bugs found**
- **Planner or designer**
  - **Author from a previous step in the software lifecycle**
- **User representative**
  - **To relate the product to what the user wants**
- **Peers of the author**
  - **Perhaps more experienced, perhaps less**
- **Apprentice**
  - **An observer who is there mostly to learn**

Fagan Inspection Process (5 steps)



COMPUTER SCIENCE

Fagan Inspection Process

- Planning ← moderator
  - Gather materials and insure that they meet entry criteria
  - Arrange for participants,
    - assign them roles,
    - insure their training
  - Arrange meeting
- Overview ← author(s)
  - explain content to the inspectors
- Preparation
  - Participants study material
- Inspection
  - Find/Report faults (Do not discuss alternative solutions)
- Rework
  - Author fixes all faults
- Follow-Up
  - Team certifies faults fixed and no new faults introduced

COMPUTER SCIENCE

Fagan Inspection

- General guidelines
  - Distribute material ahead of time
  - Use a written checklist of what should be considered
    - e.g., functional testing guidelines
  - Criticize product, not the author

UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

COMPUTER SCIENCE

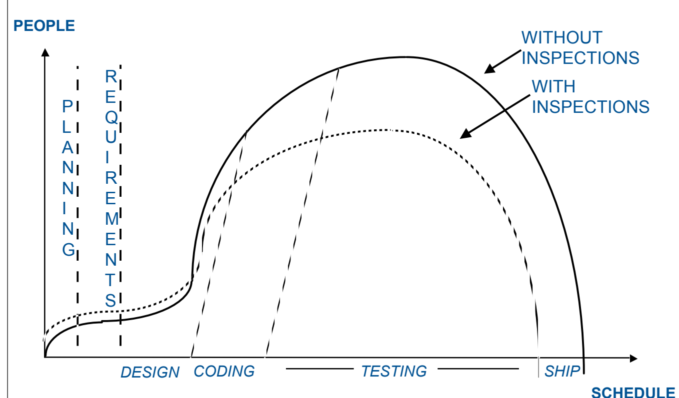
Experimental Results

- using software inspections has repeatedly been shown to be cost effective
- increases front-end costs
  - ~15% increase to development cost
- decreases overall cost
- IBM study
  - doubled number of lines of code produced per person
    - some of this due to inspection process
  - reduced faults by 2/3
  - found 60-90% of the faults
  - found faults close to when they are introduced
    - helps reduce cost

UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

COMPUTER SCIENCE

People Resource vs. Schedule



UNIVERSITY OF MASSACHUSETTS AMHERST · DEPARTMENT OF COMPUTER SCIENCE · CMPSCI 520/620 FALL 2003

Why are inspections effective

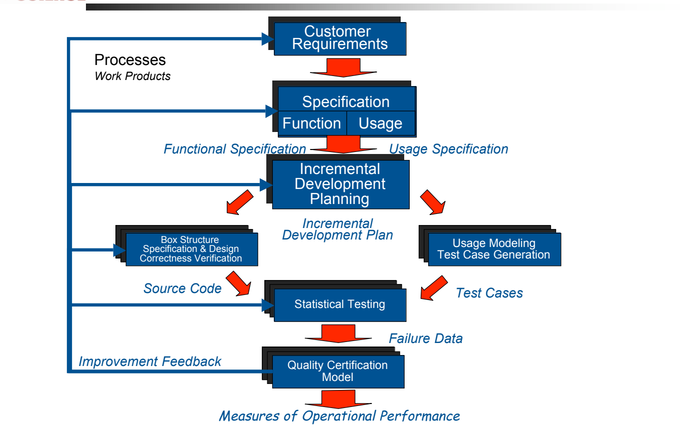
- knowing the product will be scrutinized causes developers to produce a better product
- having others scrutinize a product increases the probability that faults will be found
- walkthroughs and reviews are not as formal as inspections, but appear to also be effective
  - hard to get empirical results

What are the deficiencies?

- focus on error detection
  - what about other "ilities" -- maintenance, portability, etc.
- not applied consistently & rigorously
  - inspection shows statistical improvement, but cannot ensure quality
  - inspection should have the same results without regard to the product to which it is applied or the inspection team
- range of errors not addressed
  - team expertise limited
  - one property may have many error modalities
- human intensive and often makes ineffective use of human resources
  - e.g., skilled software engineer reviewing coding standards, comments spelling, etc.
- no automated support
  - again inefficient of human resources
- aspects of review not used appropriately
  - e.g., in Fagan process, overview often covers what should be described if documentation is adequate

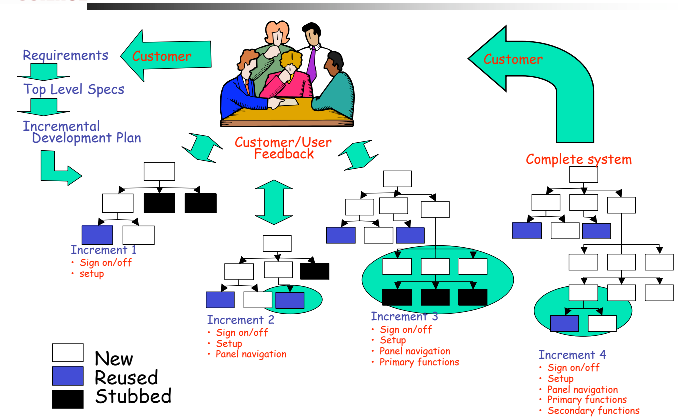
University of Massachusetts Amherst · Department of Computer Science · CMPSci 520/620 Fall 2003

Cleanroom



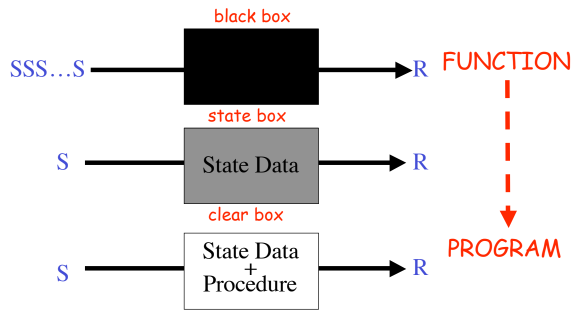
University of Massachusetts Amherst · Department of Computer Science · CMPSci 520/620 Fall 2003

Incremental development of a small system



University of Massachusetts Amherst · Department of Computer Science · CMPSci 520/620 Fall 2003

Box structure method



Verification as Review Process

- team verification of correctness takes the place of individual unit testing
  - team applies a set of correctness questions
  - correctness is established by group consensus if it is obvious
  - by formal proof techniques if it is not.
- benefits
  - intellectual control of the process
  - motivates developers to deliver error-free code
  - verification is a form of peer review
  - each person assumes responsibility for and derives a sense of ownership in the evolving product
- every person must agree that the work is correct before it is accepted -> successes are ultimately team successes, and failures are team failures.

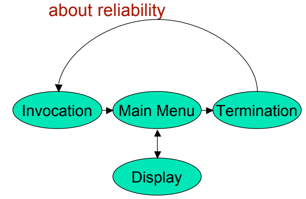
```
[f]
do
    [g]
    [h]
od
```

For all inputs, does [g] followed by [h] do [f]?

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Markov Analysis

- Factors
  - number of statistically typical (i.e., likely) usage paths through the software
  - long-run occupancy (i.e., percentage of total usage time) in each state
  - expected number of events in a test case
  - expected number of test cases before a given usage state occurs
  - expected number events between any two states
  - expected minimum number of test cases required to cover all states in the model
  - expected minimum number of test cases required to cover all transitions in the model
- Steps
  - prune the specification,
  - gauge complexity,
  - focus verification efforts,
  - identify the likelihood of given events,
  - project the test schedule, and
  - ascertain the (affordable) upper bound on inferences about reliability



University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Generation of Test Cases

- usage model->test cases
  - may be automatically generated.
- each test case is a random walk through the usage model
  - invocation->termination
- test cases constitute a "script" for use in testing
  - may be applied by human testers, or used as input to an automated test tool.
- Stopping Criterion for Testing
  - goals (e.g., target level of estimated reliability) are achieved
  - or quality standards (e.g., errors/KLOC) are violated
- Statistical Hypothesis Testing

Confidence level (%)

Reliability level (*r*)

r \ %	90	95	99	99.9
0.9	22	29	44	66
0.95	45	59	90	135
0.99	230	299	459	688
0.999	2302	2993	4603	6905

University of Massachusetts Amherst · Department of Computer Science · CmpSci 520/620 Fall 2003

Software Metrics

- measures that predict qualities about software
- can be applied to any of the products (e.g., design, code, test cases) or to the process (e.g., Capability Maturity Model)
- Qualities measured by software metrics
  - performance
  - user-friendliness
  - resources
    - memory/storage
    - development costs
    - maintenance cost
  - quality
    - maintainabity
    - reliability
    - completeness
    - consistency
    - complexity

Function Points

- proposed by Albrecht in 1979
  - Originally applied to code
- UFP =
  number of inputs x w1 +
  number of outputs x w2 +
  number of user inquiries x w3
  number of files x w4 +
  number of external references x w5
- function points = UFP* TCA = UFP* (.65 + 0.01 * SUM(Fi))
  - where the degree of influence, DI= SUM(Fi) is the sum of complexity adjustment values, Fi
- metrics:
  - productivity: FP/person-month
  - quality: defects/FP
  - cost: $/FP

- weights:

	Simple	Average	Complex
w1	3	4	6
w2	3	5	7
w3	3	4	6
w4	7	10	15
w5	5	7	10

More Quality Metrics

- Modularity
  - cohesion metric
    - applied to unit design
    - the relationship among the elements of a module
    - best cohesion level is functional, and the worst is coincidental.
  - Cruickshank and Gaffney Cohesion Strength
    Strength = $\sqrt{X^2 + Y^2}$
    - where:
    - X = reciprocal of the number of assignment statements in the module
    - Y = number of unique function outputs divided by number of unique function inputs

More Quality Metrics

- Modularity
  - coupling
    - applied to system and unit designs
    - measure of the degree to which modules share data
    - data coupling (the sharing of data via parameter lists) is the best type of coupling, while common coupling (the sharing of data via global or common areas) is the worst.
    - a lower coupling value is better.
  - Cruickshank and Gaffney Coupling:
    - M_j = sum of the number of input and output items shared between components i & j
    - Z_i = average number of input and output items shared over m components with component i
    - n = number of components in the software product

$$Coupling = \frac{\sum_{i=1}^{n} Z_i}{n}$$

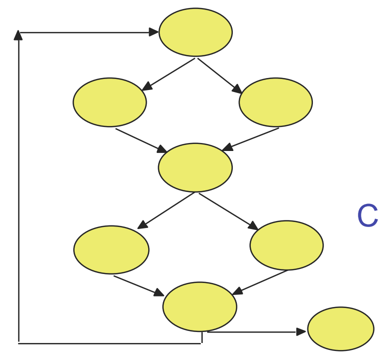
where:

$$Z_i = \frac{\sum_{j=1}^{m} M_j}{m}$$

McCabe's cyclomatic complexity

- Complexity measured by control flow information
  - based on a control flow graph where e is number of edges, n is number of nodes, p is number of connected components
- McCabe's Cyclomatic Complexity:
  - v = e - n + 2
    - where:
    - v = complexity of the graph
    - e = number of edges (program flows between nodes)
    - n = number of nodes (sequential groups of program statements)
  - if a strongly connected graph is constructed (one in which there is an edge between the exit node and entry node), the calculation is
  - v = e - n + 1

Example



n = 8
e = 10
p = 1

C = 10 - 8 + 2 = 4

Software Science

- Halstead applied information theory to computer science
- metrics
  - **n_1 number of distinct operators**
  - **n_2 number of distinct operands**
  - **N_1 total number of occurrences of operators**
  - **N_2 total number of occurrences of operands**
- program level estimator

$$\mathcal{D} = 1 / \mathcal{L} = (n_1 / 2)(N_2 / n_2)$$

$$\mathcal{L} = 1 / \mathcal{D} = (2/n_1)(n_2 / N_2)$$

difficulty increases as operators are introduced ($n_1/2$ increases) and as operands are used repetitively (N_2 / n_2 increases)

- programming time

$$\mathcal{T} = \mathcal{E} / \mathcal{S}$$

where $\mathcal{S}$ is the "Stroud number"

$5 \leq \mathcal{S} \leq 20$, usually 18

Software Science (continued)

- language level

$$\lambda = \mathcal{L} \times \mathcal{V}^* = \mathcal{L}^2 \mathcal{V}^*$$

$\lambda_{PL/1} = 1.53$, $\lambda_{Algol} = 1.21$,
$\lambda_{Fortran} = 1.14$, $\lambda_{CDC\ assmblr} = 0.88$

- predicted effort

$$\mathcal{E} = \mathcal{V}^{*3} / \lambda^2$$

Quality Metrics for Code

- Understandability
  - size metrics
    - lines of code
    - function points
    - function count
  - traceability metrics
    - number of comment lines per total source lines of code
    - percent comment lines of total lines
    - correctness of comments
- Predicting quality
  - LOC X domain seems to be the most reliable predictor