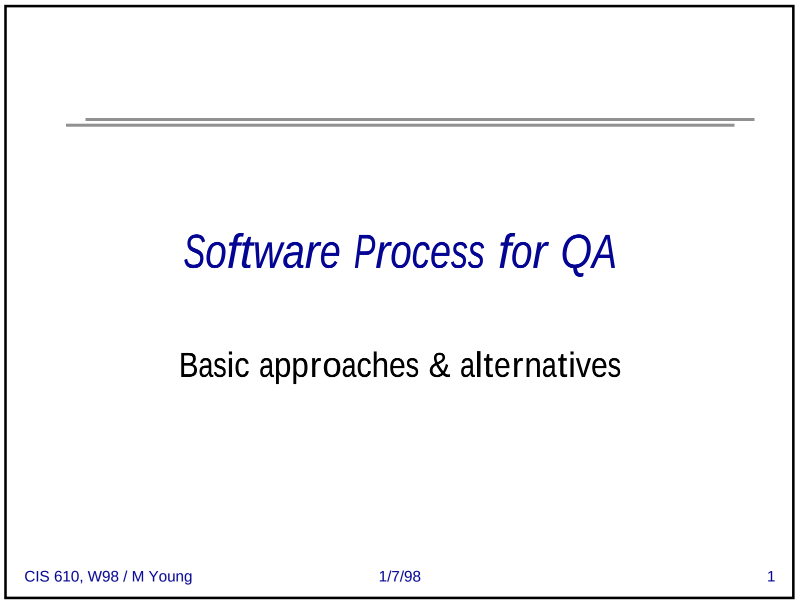
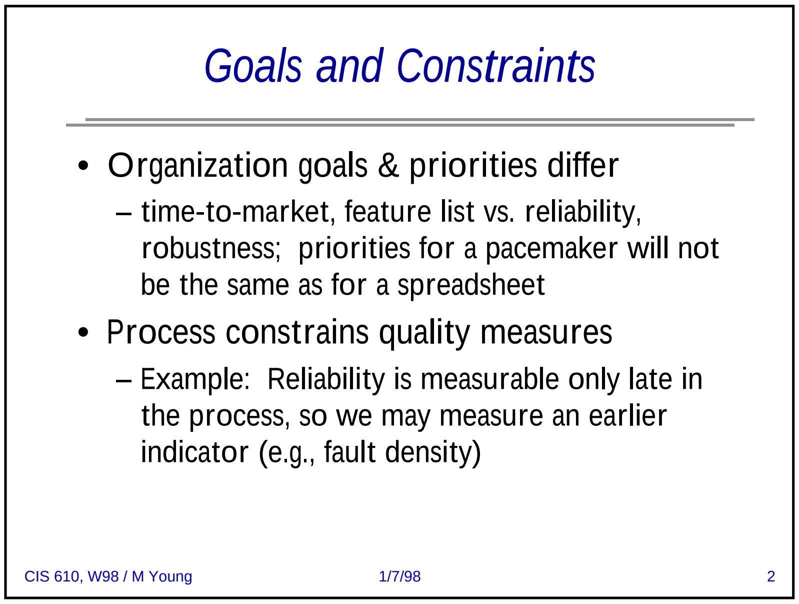


Software Process for QA

Basic approaches & alternatives

This introduction and overview is intended to provide some basic background on software process (sometimes called “lifecycles”), and to indicate some basic issues that we’ll need to think about when considering approaches to analysis and testing. We’ll consider software process issues in greater depth later, after we’ve looked more closely at particular analysis and testing techniques.



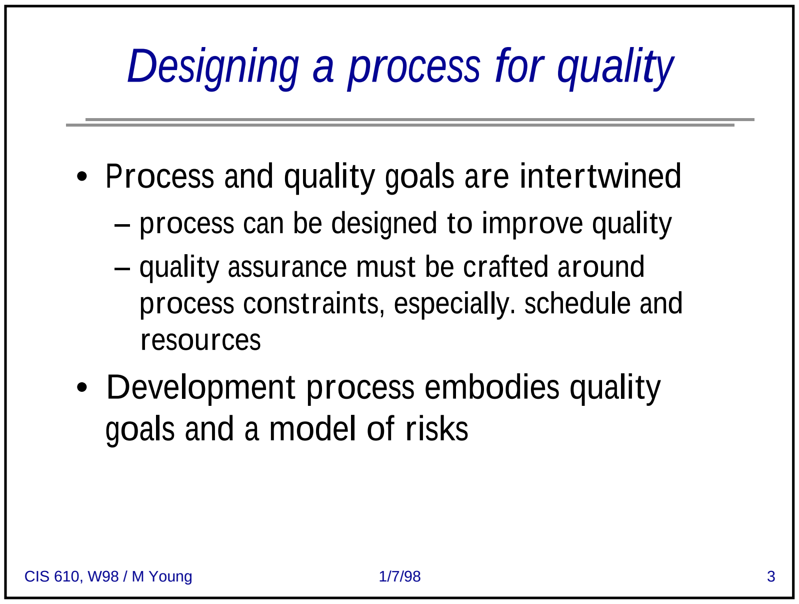
Goals and Constraints

- Organization goals & priorities differ
  - time-to-market, feature list vs. reliability, robustness; priorities for a pacemaker will not be the same as for a spreadsheet
- Process constrains quality measures
  - Example: Reliability is measurable only late in the process, so we may measure an earlier indicator (e.g., fault density)

CIS 610, W98 / M Young 1/7/98 2

The first point is obvious, but often forgotten: Different kinds of software require different approaches, and different organizations have different priorities. If we are producing a medical device like a pacemaker, dependability will (or should) be the preeminent value; in particular, it may be much more important than time-to-market. The opposite may be true for production of the next great web-page editor: If we don't get it to market before the competition, dependability won't be much of an asset unless the competing product is just unbearably bad. The process by which software is developed will reflect these priorities. It's silly to talk about the "right" way to approach software quality without considering these top-level priorities.

A process model (or "lifecycle") presents certain opportunities for quality assurance measures, and imposes certain constraints. For example, in many development processes, it is not practical to measure reliability early in the process, before there is a running system. We may choose to use another measure, like fault density, simply because it "fits" in an earlier stage (e.g., based on design and code reviews) and provides earlier useful feedback.



Designing a process for quality

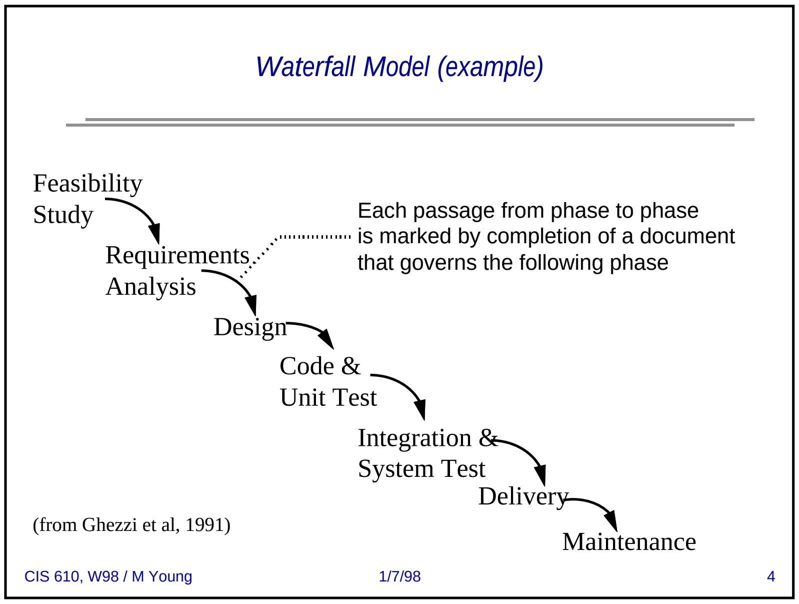
- Process and quality goals are intertwined
  - process can be designed to improve quality
  - quality assurance must be crafted around process constraints, especially. schedule and resources
- Development process embodies quality goals and a model of risks

CIS 610, W98 / M Young 1/7/98 3

The relation between process and quality assurance is not a one-way dependence. We may also structure the development process in ways that make it more practical to measure and improve dependability (as well as other software qualities, like usability and maintainability).

A development process embodies quality goals and a model of risks. In other words, when we design a software development process including quality assurance activities, we are implicitly or explicitly making assumptions about what can go wrong and how we will avoid or correct those problems.

In the next few slides, I'll try to make that more concrete by looking at a few example process models: A naive waterfall model, a more realistic waterfall model, and the "spiral" model for incremental development, along with two analysis and testing methodologies, the "cleanroom" approach developed by Harlan Mills at IBM and the SRET approach developed by John Musa at AT& T.



Waterfall Model (example)

Feasibility Study
Requirements Analysis
Design
Code & Unit Test
Integration & System Test
Delivery
Maintenance
Each passage from phase to phase is marked by completion of a document that governs the following phase
(from Ghezzi et al, 1991)

CIS 610, W98 / M Young 1/7/98 4

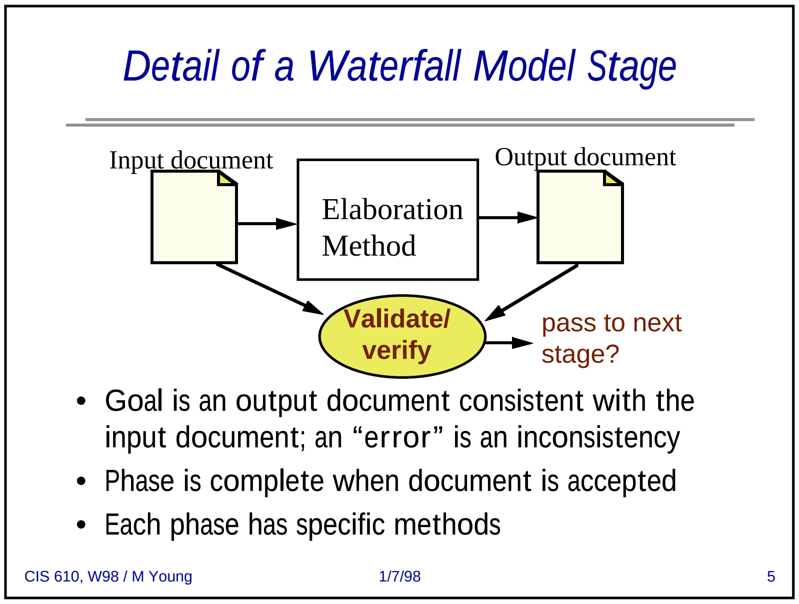
In its simplest (naive) form, the waterfall model consists of a set of development stages in which one document is used as the basis for the construction of another. The requirements analysis phase, for example, produces a requirements specification which serves as the basis for producing design documents. The particular stages vary, and they can be divided up in different ways; e.g., the “feasibility analysis” stage isn’t always present, “design” is often split at least into “high level design” and “detailed design;” also “detailed design” is often merged with “code and unit test.” The important characteristics of this model are that

 * There is a sequence of well-defined stages, and a project moves discretely from stage to stage (i.e., there are “milestones” at which one stage is considered closed and another begins).

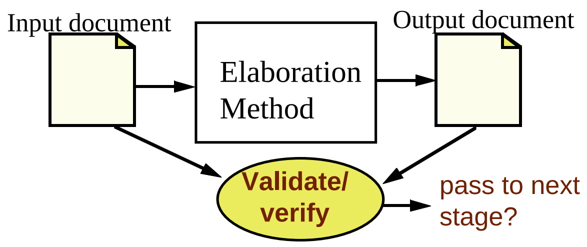
* The input of each stage is a document or set of documents from the prior stage, and its output is a set of documents to be used in the following stage. (We consider code to be a “document” in this sense.) (The earliest and latest stages are special cases --- requirements analysis may have no input document, and maintenance modifies existing documents rather than producing a new document.)

* The “acceptance criterion” for completing a stage is that the output documents are consistent with the input documents.

There are several problems with this naive version of the waterfall model, and in fact few software development processes are really this simple-minded. We’ll look at a more realistic version of the waterfall model in a bit.



Detail of a Waterfall Model Stage

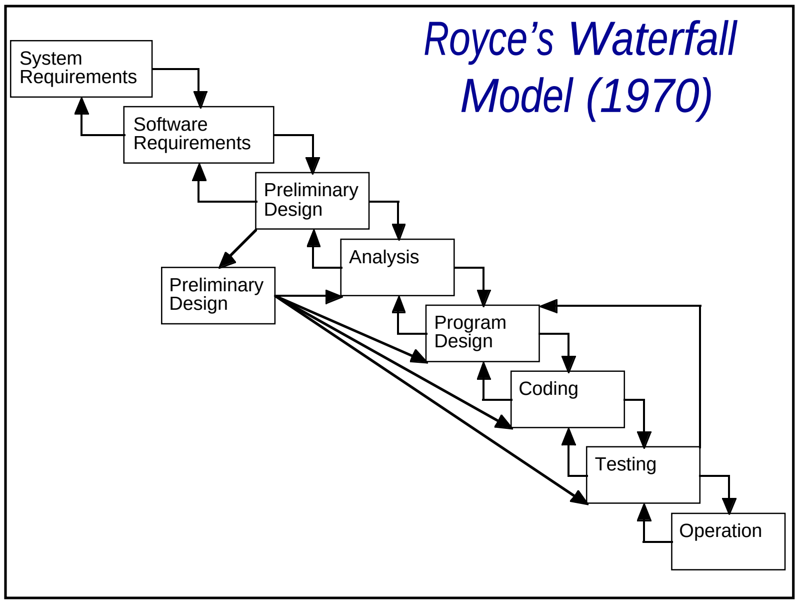


- Goal is an output document consistent with the input document; an "error" is an inconsistency
- Phase is complete when document is accepted
- Each phase has specific methods

Here is a more detailed view of an individual stage in a waterfall model. There is some input document (e.g., a design), some output document (e.g., code), and some method of getting from the input to the output. The elaboration method may be as simple as "write code" or as complex as an object-oriented modeling and design methodology. The elaboration method varies from stage to stage. For example, there may be one method for requirements elicitation, another for high-level design, another for detailed design, and another for coding.

A set of document notations and elaboration methods designed to work together is often called a "development methodology," e.g., object-oriented analysis and design using UML.

In addition to an elaboration method, each stage is usually associated with a method for validation or verification. For example, a requirements specification may be validated by user walk-throughs and acceptance, design may be verified by walk-throughs or inspections, and code may be verified by inspections and unit testing. Milestones in the process, including completion of a stage, are marked by successfully passing the validation or verification checks for consistency.



Royce's Waterfall Model (circa 1970)

Here is a more realistic version of the waterfall model. Note that it is not recent --- the naive waterfall model was never really the way development was carried out.

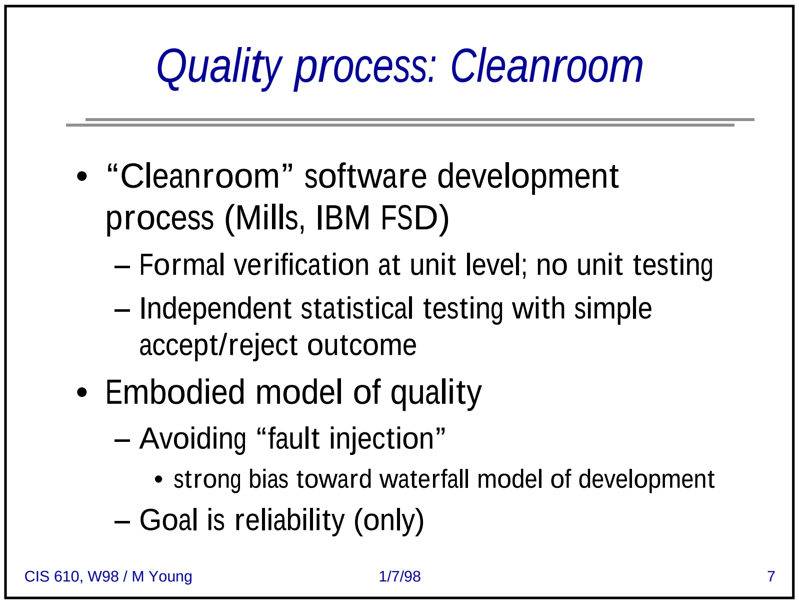
W. Royce presented a process model summarized in the diagram above at the International Conference on Software Engineering in 1970. Royce was at TRW, an Aerospace company, and the process model is influenced both by the constraints of that domain (e.g., the importance of performance analysis using prototypes) and by the customary development processes for systems in that domain. Also, since the domain was "embedded" applications in which software was one part of a larger system, the software requirements and design follow overall system requirements and design.

The "analysis" phase in Royce's model seems to be in a peculiar place, but that is because today the term "analysis" is used in a different sense. For Royce, it is analysis of characteristics of a system design, including performance.

Note that it Royce's model is significantly more complex than the simple waterfall model that is still commonly used in many organizations, and it contains many "modern" features:

* There is explicit feedback from each stage to the prior stage. Royce recognized that one could not "freeze" requirements before entering design, design before code, etc., but that there would always be some iteration. Note that testing feeds back beyond coding to program design.

* Prototyping plays a key role, and feeds most of the later stages.



Quality process: Cleanroom

- "Cleanroom" software development process (Mills, IBM FSD)
  - Formal verification at unit level; no unit testing
  - Independent statistical testing with simple accept/reject outcome
- Embodied model of quality
  - Avoiding "fault injection"
    - strong bias toward waterfall model of development
  - Goal is reliability (only)

CIS 610, W98 / M Young 1/7/98 7

Cleanroom is an example of a (very controversial) quality assurance methodology which prescribes particular analysis and test methods at particular points in a development process.

Cleanroom is based on a model of avoiding "fault injection." The model can be paraphrased (some might say caricatured) as follows:

* The goal is to produce reliable software, i.e., the relevant measure is mean time to failure (or some equally quantifiable variation). [But there are ways of weighting "critical" failures vs. annoyances, and so on.]

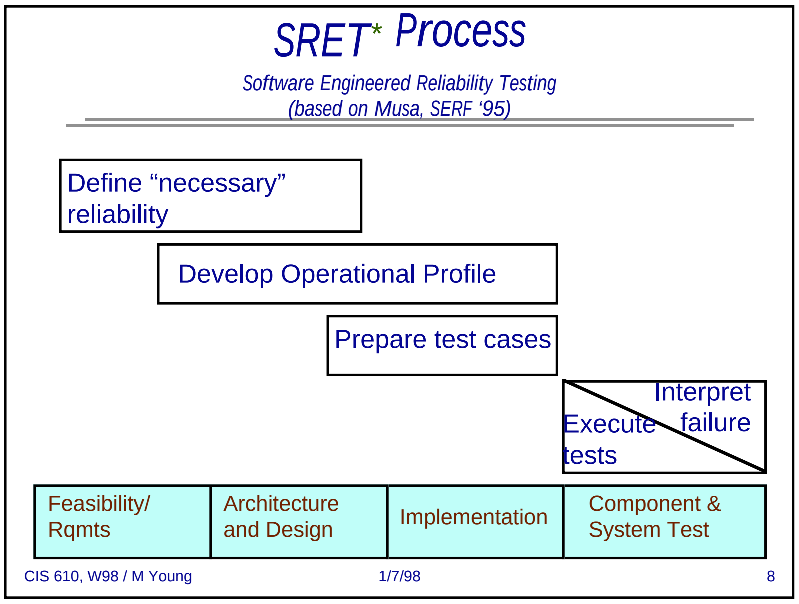
* The goal is to be reached by avoiding faults, rather than correcting them. The purpose of testing is measurement, not debugging, and the outcome is a reliability estimate that is either acceptable or not.

Based on this model, the Cleanroom methodology prescribes:

* No unit testing. In fact, programmers are not permitted to run their code. The only methods to be used at the unit level are formal verification and inspections.

* System testing is performed by an independent test group, not by developers. It is based on specifications and a model of use, rather than the structure of the software (i.e., it is a "black box") method. Extensive random testing is used to obtain a reliability estimate, on which an accept/reject decision is based.

Strong claims have been made for the Cleanroom approach, but those claims (and the methodology) are not widely accepted. We won't evaluate it in more detail at this point; rather, the interesting point is the way the analysis and testing techniques are tied to the development process.



SRET* Process

Software Engineered Reliability Testing (based on Musa, SERF '95)

CIS 610, W98 / M Young 1/7/98 8

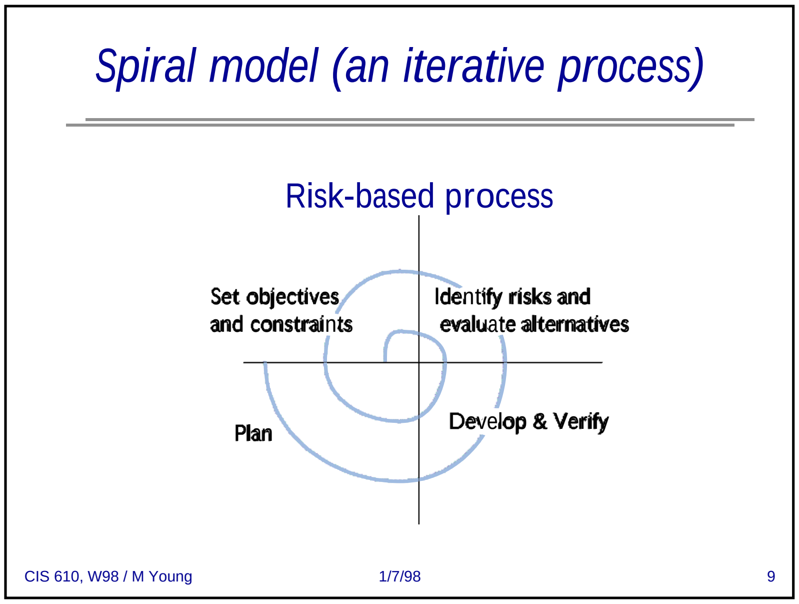
The Software Reliability Engineered Testing (SRET) process was developed primarily by John Musa when he was at AT&T. Like Cleanroom, it is a methodology for systems with high reliability requirements. (Cleanroom was developed at IBM Federal Systems Division, which among other things wrote space shuttle software, and SRET was developed at AT&T where telephone switching systems were developed.) Like Cleanroom, SRET aims to produce a reliability measure for software, but it ties testing much more closely to design and implementation.

SRET is interesting in several regards:

* There is a series of test development stages coordinated with overall project development stages (following roughly a waterfall model).

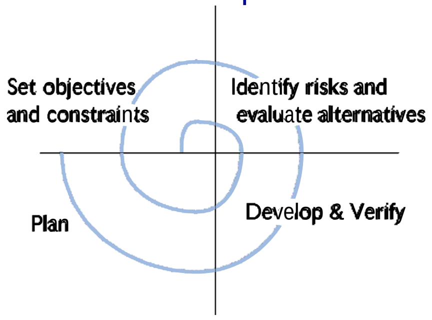
* Test development is not black-box; test case design is based partly on the system architecture and design. Still, development of an operational profile to enable reliability estimation in system test is a key part of the method.

* SRET includes *process feedback*. Interpretation of a failure is used not only for debugging the current product, but for building a base of experience for future testing and development. A failure "interpretation" includes an analysis of how the fault was originally introduced, and how it could have been prevented or detected as early as possible.



Spiral model (an iterative process)

Risk-based process



CIS 610, W98 / M Young 1/7/98 9

The waterfall model, in either its naive or more realistic forms, is basically a process for developing a single product from scratch. There are many "iterative" process models that accommodate prototyping, multiple versions and revisions of a product, and/or development of several related products. The spiral model, introduced by Barry Boehm around 1988, describes a family of models in which iteration is based primarily on identification and control of risk. (Like the waterfall model, the spiral model has many variations.)

In incremental development, one produces many prototype versions of the system, starting very simply and working toward the complete target system. The key idea of the spiral model is that each version (until a "final" version delivered to the user) is intended to provide *information*, and that the information is designed to reduce risk. For example, if the most important risk is a product that is not usable, the first prototype may be aimed at usability testing. If performance is judged a high risk, then an early prototype may be aimed at testing performance-critical portions of the system. (Contrast this with phased projects whose main goal is early delivery, in which low-risk deliverables are produced first, saving the higher-risk features until later.)

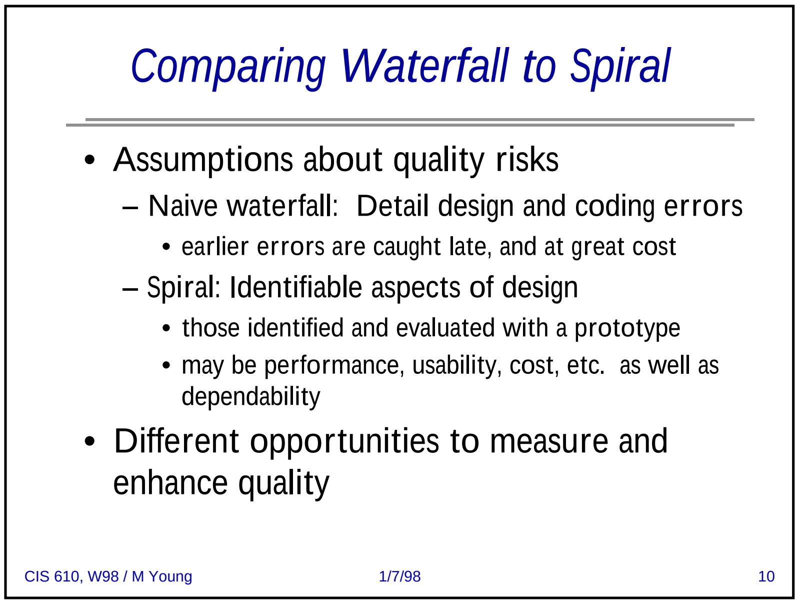
Each prototype is one "turn of the spiral," and is divided into phases:

* Set objectives and constraints for the product and project.

* Identify risks and evaluate alternatives. The purpose of one turn of the spiral is to provide more information about particular, explicitly identified risks, and to provide the necessary information to make an informed choice among alternatives. (One alternative may be "abandon the project.")

* Develop and verify. The verification methods are (obviously?) chosen to provide information relevant to the identified risks.

* Plan the next version of the project.



Comparing Waterfall to Spiral

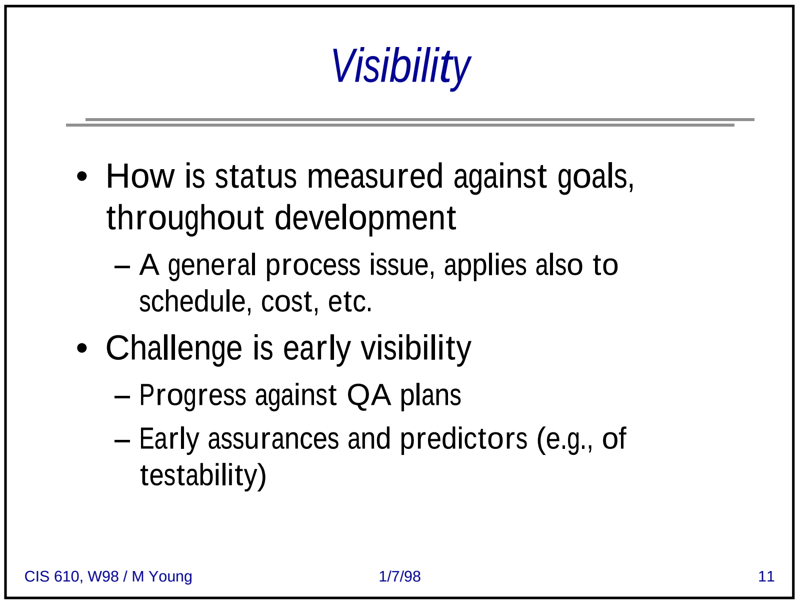
- Assumptions about quality risks
  - Naive waterfall: Detail design and coding errors
    - earlier errors are caught late, and at great cost
  - Spiral: Identifiable aspects of design
    - those identified and evaluated with a prototype
    - may be performance, usability, cost, etc. as well as dependability
- Different opportunities to measure and enhance quality

CIS 610, W98 / M Young 1/7/98 10

It is interesting to compare waterfall models to spiral models in terms of their assumptions about quality risks, and the opportunities they afford for analysis and testing.

The naive waterfall model is based on establishing consistency between a series of documents. One of the biggest problems of waterfall models is that errors which are not caught early can “propagate through” to the final system, where they are very expensive to correct. In particular, user acceptance of a requirements document does not necessarily imply that users will like the finished system; validation (vs. verification) at only the earliest and latest stages make waterfall methods vulnerable to changing and incompletely understood requirements.

The spiral model, or any model with extensive prototyping, can avoid some of the problems of the waterfall model. A main assumption (and vulnerability) of the spiral model, on the other hand, is that risks can be accurately identified and assessed in advance. If the choices of what to prototype, and how to evaluate the prototype, are not made well, the spiral model can deteriorate into a “hack and fix” model of development.



Visibility

- How is status measured against goals, throughout development
  - A general process issue, applies also to schedule, cost, etc.
- Challenge is early visibility
  - Progress against QA plans
  - Early assurances and predictors (e.g., of testability)

CIS 610, W98 / M Young 1/7/98 11

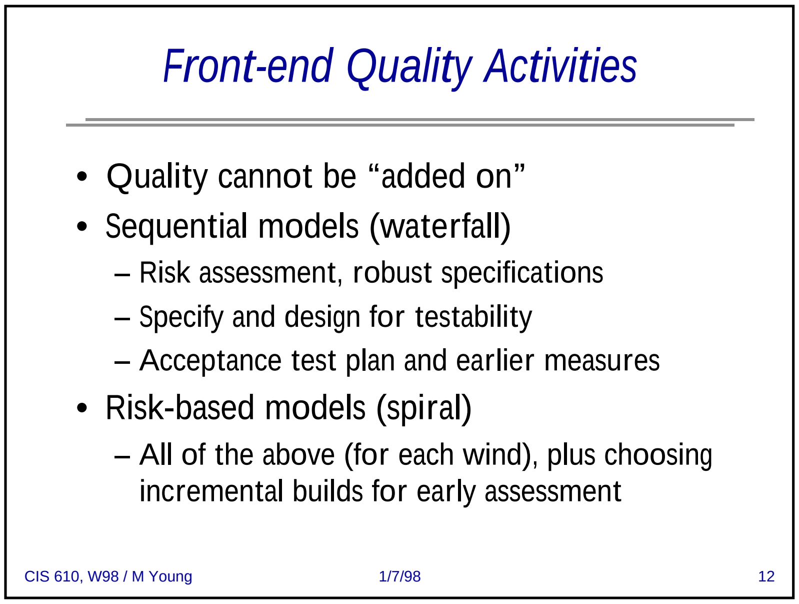
A major criterion for any process is to achieve *visibility*, which means the ability to measure progress against goals. For example, schedule visibility means the ability to determine whether a project is on schedule or, if not, how far it is behind or ahead. The need for visibility applies to quality assurance as well: How can we tell whether we are producing a product that meets our goals for quality?

It is not too hard to come up with ways of making quality visible eventually. For example, we could measure the number of problem reports phoned in by users. That is like measuring the schedule only by the delivery date; by the time we recognize a problem, it is too late to correct. The challenge, for quality as for schedule and cost, is early visibility.

Schedule visibility is often approached by setting a number of milestones: By June 10, this is what we shall have accomplished, etc. Progress against plans can be measured for test and analysis as well (as in the SRET process), although by itself this does not say very much about the quality of the product eventually produced.

Another approach is to employ early predictors of quality, such as fault density measures based on problems found in inspections and unit testing. These measures are difficult to interpret by themselves, but they can be useful if there is a base of historical data to compare to (and if the current project is sufficiently similar to prior projects).

In addition to early indicators of product quality, it is useful to have early indicators of the analysis and test effort required. A system can be designed in ways that make it more or less expensive to test, and it is useful to use testability as an important criterion in evaluating design alternatives.



Front-end Quality Activities

- Quality cannot be “added on”
- Sequential models (waterfall)
  - – Risk assessment, robust specifications
  - – Specify and design for testability
  - – Acceptance test plan and earlier measures
- Risk-based models (spiral)
  - – All of the above (for each wind), plus choosing incremental builds for early assessment

CIS 610, W98 / M Young 1/7/98 12

It is a huge mistake to view testing as a phase that converts a poor quality product to a high quality product. Analysis and testing activities must go on at every point in development. (This is equally true of other kinds of quality assurance, e.g., usability and maintainability; none of them are qualities that can be “added on” at the end of a project.)

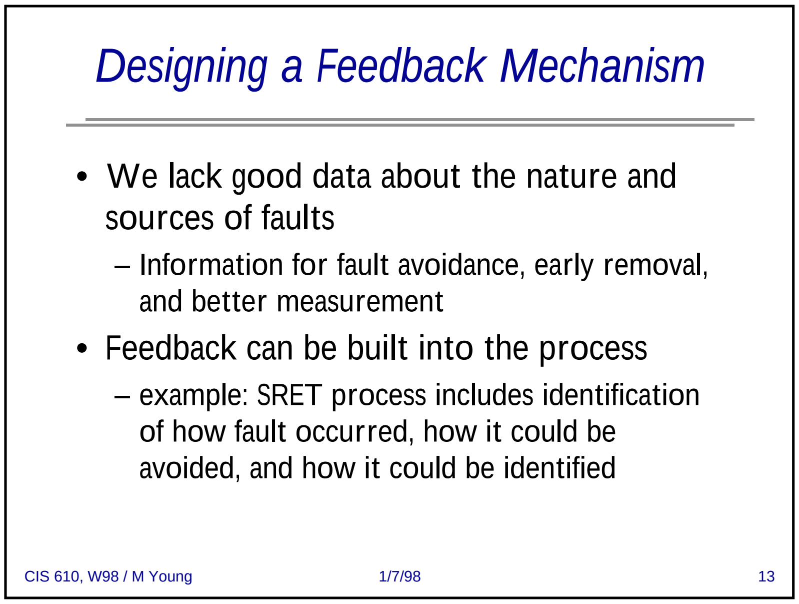
In sequential (waterfall-like) models, some early activities include:

* Risk assessment: Identifying as soon as possible what kinds of problems might be anticipated, and their relative importance.

* Robust specification: Making sure the specification is complete (including, for example, desired responses to undesirable situations), and that it is stated in ways that are precise verifiable. For example, a specification that a system “responds quickly” is useless for verification, but a specification that it “responds within 1.5 seconds” can be verified.

* In requirements analysis, it is useful to associate specific acceptance conditions with each user requirement (like the response time requirement above), and to produce an acceptance test plan as part of the requirements document.

Somewhat surprisingly, perhaps, development models with extensive prototyping can be more challenging for early assessment. In principle, the early activities of sequential models can be performed for each prototype, but in practice it is difficult (and perhaps unreasonable) to perform rigorous quality control for prototypes. Prototyping is pointless, after all, if it takes as long to built a prototype as to build the final system; something has to give, and frequently quality control is part of that. This is particularly a problem when prototype code and is reused in the delivered product. One opportunity that incremental development affords is that the riskiest parts of a system may be built early, providing an extended opportunity for assessment.



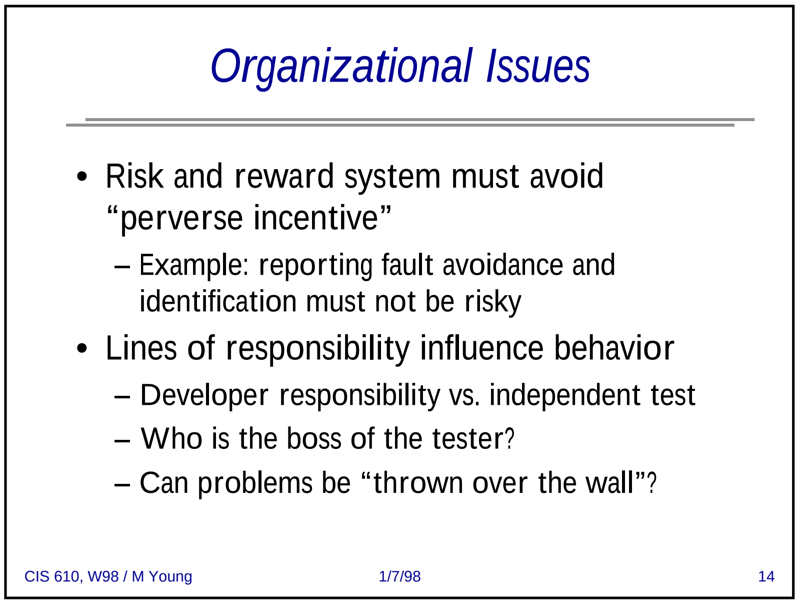
Designing a Feedback Mechanism

- We lack good data about the nature and sources of faults
  - Information for fault avoidance, early removal, and better measurement
- Feedback can be built into the process
  - example: SRET process includes identification of how fault occurred, how it could be avoided, and how it could be identified

CIS 610, W98 / M Young 1/7/98 13

One of the big advantages of hardware designers over software designers is that they have very good models of the kinds of faults that occur in manufacturing, and the kinds of failures that happen in the field. Semiconductor tests look for “stuck-at” faults, because these are known to be one of the most common manufacturing faults. Aircraft inspectors look for stress fractures in aircraft bodies, especially in older aircraft, based on a history of aviation accidents. When an aircraft fails, an extensive failure analysis is carried out, and the results are used to refine aircraft inspection procedures.

Software developers lack well-developed fault models (although this is starting to change). The lack of fault models is partly because faults depend so much on context: What kind of software, what programming language, what kind of development organization and methodology? Feedback mechanisms, such as the failure analysis part of SRET, are a way of building up a fault model that works in a particular context. As we will see when we look at software inspections, feedback and the compilation of checklists is also a key ingredient in inspections.



Organizational Issues

- Risk and reward system must avoid "perverse incentive"
  - Example: reporting fault avoidance and identification must not be risky
- Lines of responsibility influence behavior
  - Developer responsibility vs. independent test
  - Who is the boss of the tester?
  - Can problems be "thrown over the wall"?

CIS 610, W98 / M Young 1/7/98 14

Finally, organizational structures are closely tied to processes, and can have positive or negative effects on quality assurance.

One problem in process design and organization structure is "perverse incentives," or "the law of unintended consequences." A classic example is evaluating programmer productivity by lines of code written — a sure way to prevent reuse and tight, clean code. With respect to quality, it is important to ensure that the incentive structure rewards prevention and early detection of problems. For example, a failure analysis activity as in SRET could fail if it requires programmers to "take the blame" for faults.

One of the reasons that many organizations have a separate quality assurance function (either a separate testing team, or testing specialists within the development team) is to make sure that testers have an incentive to find faults, and no incentive to let them pass.

Lines of authority also influence the incentive structure. For example, if the boss of the tester is under deadline pressure to release a product, there may be an incentive to "lower the barrier;" on the other hand, if quality assurance is completely free of schedule pressures, they could make make meeting deadlines difficult.

In some organizations, there is a point where responsibility for a product passes from one team (e.g., a development group) to another (a maintainence group). If poorly managed, this transition creates a perverse incentive to "throw a problem over the wall," hiding a problem or its severity long enough to make it someone else's problem.