Requirements Elicitation

Proactive vs. Reactive Elicitation

- Users seldom provide complete, reasonable requirements without coaxing.
  - The user doesn't know what is practical or possible.
- Requirements elicitation is an active process
  - gathering information
  - negotiating
    - We could do X, but it would take Y months longer.
  - suggesting alternatives

Problems vs. Solutions

- Users typically have a solution in mind, and it is typically a small variation on current activities.
- *Back up. Understand the problem.*
- Separate the *what* from the *how*
  - The *how* is already on your mind, but it must be carefully partitioned from the *what*.

Who do you talk to?

If the client is an organization, analysts should consult with

- Someone with authority
  - ensure an organizational commitment (“buy-in”) to the project objectives and direction
- Each user group
  - *at all levels: the boss may not know how it's* really *done*
- Each enabling group
  - unhappy people can ensure failure

Organizational Context

- Elicitation problems depend partly on the organizational context of system development
- Example contexts and variations:
  - Central development organization vs. decentralized development
  - Client/Buyer vs. Market
- Sometimes we can adjust the context; more often we must adapt to it

External Clients & Contract Projects

- Advantages
  - Variable resource levels and kinds
  - Less fixed budget commitment
  - “Flatter” organizations
- Problems
  - Premature specification freezing
  - Institutional memory and relationships
  - Products vs. product lines

Specifications as Contracts

- Problem: Premature specification freeze
  - May narrow solution space and stifle creative approaches
  - Changes may become very expensive
  - Works best when developers produce a product line with limited variatons (“precedented” products)
- Problem: Product lines
  - Contracting rules can discourage reuse and infrastructure development
    - But some contract developers do well by amortizing development across several clients

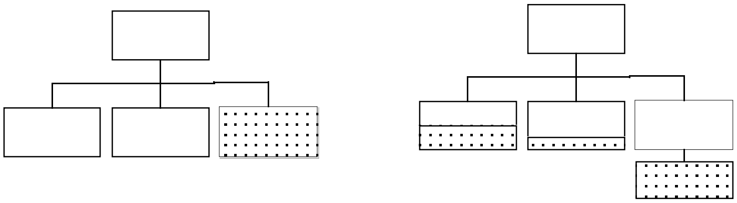
Developing for a Market

e.g., shrink-wrap software

- The “client” is potential buyers in a software market, but we still need requirements analysis
- Approaches:
  - Study the competition and market
    - and talk to users of the competing or related products
  - Recruit potential users
    - surveys, interviews, mock-ups
    - the “client” may need to be paid!
  - Prototypes and incremental deliveries

Internal Development: Centralized or Decentralized?

Organizational context affects requirements analysis



- In a large enterprise, developers can be organized in a single centralized “service” organization, or small development organizations can be distributed throughout the enterprise

Internal Development: Centralized vs. Decentralized

Software system development for clients within the same enterprise (e.g., company or agency)

- Centralized resource
  - Serves clients in many sub-areas of the enterprise
  - Clients are in competition for the resource
- Decentralized resource
  - Developers are distributed throughout the enterprise
  - Clients have dedicated resource

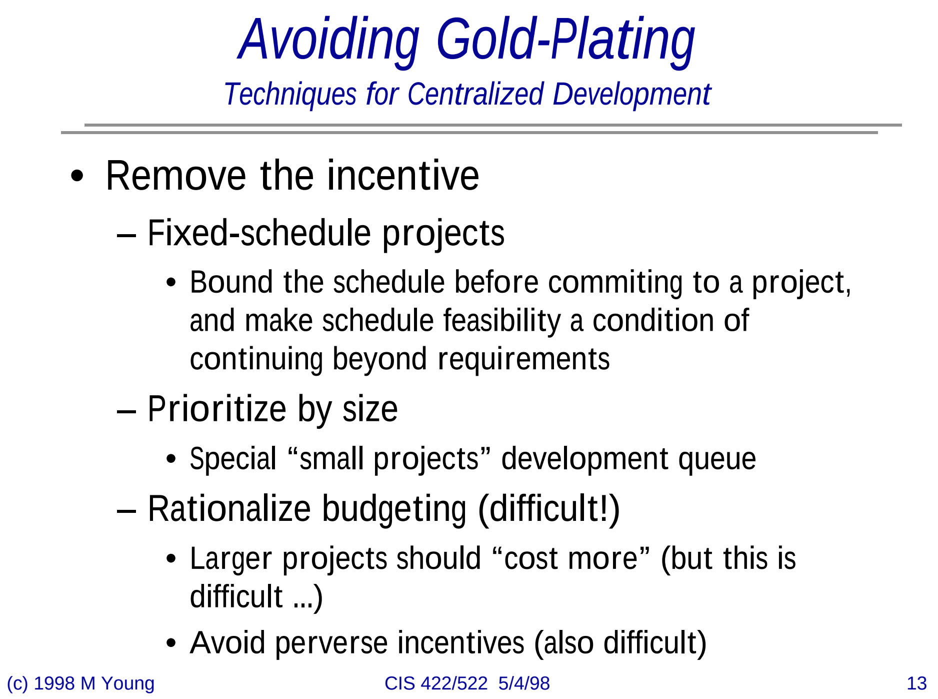
Requirements Elicitation in Centralized Development

- Advantages:
  - Larger development organization with more specialized work roles. Experienced analysts work with a variety of clients and apply “tried and true” approaches
- Problems
  - Developers lack domain expertise
  - “Gold plating”: Competition for development resource encourages clients to hold resource as long as possible

Developing Domain Expertise

Techniques for Centralized Development

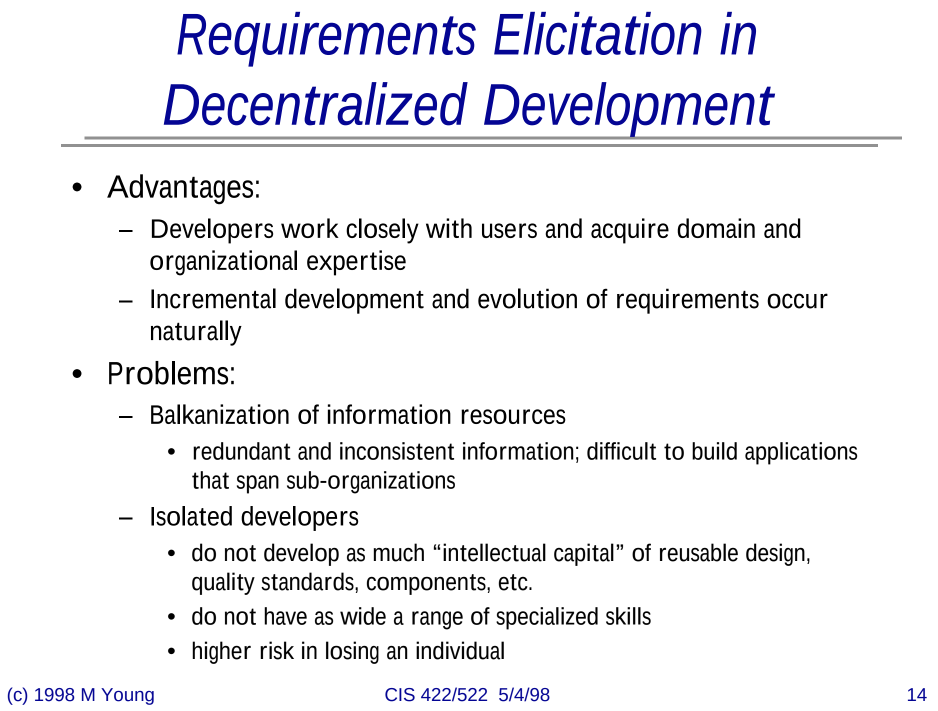
- Explicitly schedule and budget for domain analysis and training
- Develop specializations within the development organization
  - but also cross-train to spread the knowledge



Avoiding Gold-Plating

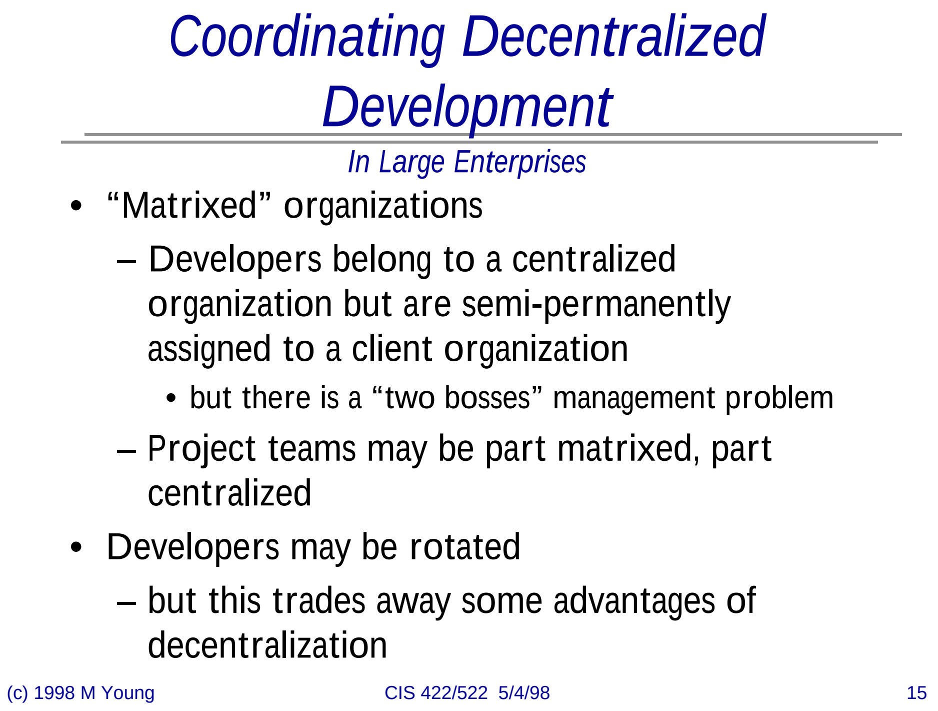
Techniques for Centralized Development

- Remove the incentive
  - – Fixed-schedule projects
    - Bound the schedule before commiting to a project, and make schedule feasibility a condition of continuing beyond requirements
  - – Prioritize by size
    - Special “small projects” development queue
  - – Rationalize budgeting (difficult!)
    - Larger projects should “cost more” (but this is difficult ...)
    - Avoid perverse incentives (also difficult)



Requirements Elicitation in Decentralized Development

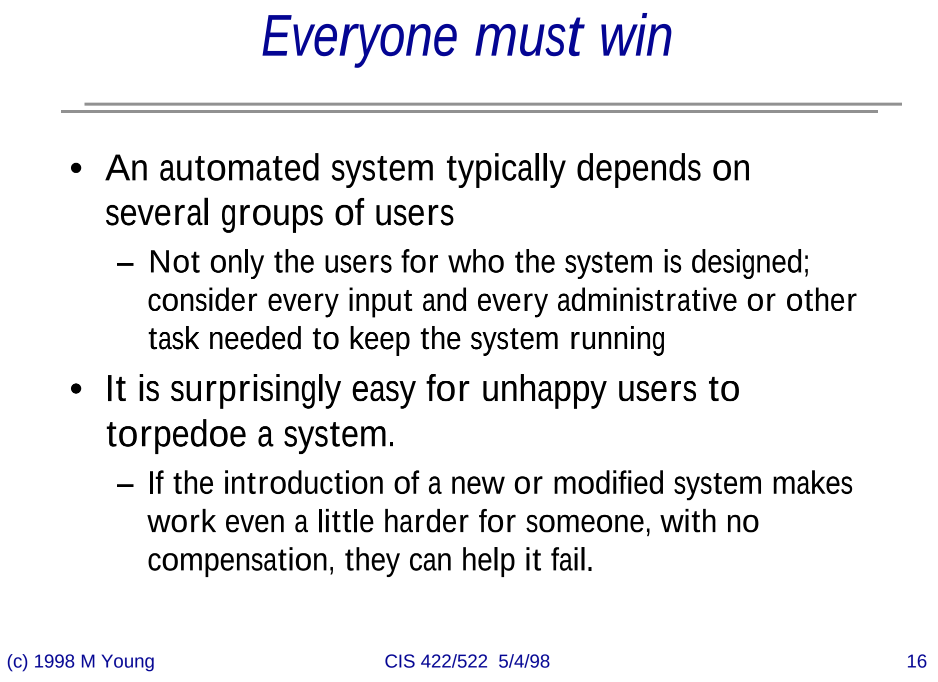
- Advantages:
  - – Developers work closely with users and acquire domain and organizational expertise
  - – Incremental development and evolution of requirements occur naturally
- Problems:
  - – Balkanization of information resources
    - redundant and inconsistent information; difficult to build applications that span sub-organizations
  - – Isolated developers
    - do not develop as much “intellectual capital” of reusable design, quality standards, components, etc.
    - do not have as wide a range of specialized skills
    - higher risk in losing an individual



Coordinating Decentralized Development

In Large Enterprises

- "Matrixed" organizations
  - – Developers belong to a centralized organization but are semi-permanently assigned to a client organization
    - but there is a "two bosses" management problem
  - – Project teams may be part matrixed, part centralized
- Developers may be rotated
  - – but this trades away some advantages of decentralization



Everyone must win

- An automated system typically depends on several groups of users
  - – Not only the users for who the system is designed; consider every input and every administrative or other task needed to keep the system running
- It is surprisingly easy for unhappy users to torpedoe a system.
  - – If the introduction of a new or modified system makes work even a little harder for someone, with no compensation, they can help it fail.

A Failure to Provide Win Conditions

City of Eugene, Oregon, information system to schedule public works projects (repairing signs, patching roads, trimming trees), early 1980s

- *Inputs:* Inspectors fill out forms describing needed repairs.
- *Outputs:* Planning reports for managers

DISASTER: *No win condition for inspectors. The system was technically sound, but failed miserably.*

Lollipops

- After the doctor gives the child a shot, she also gives him a candy

➠ Try to ensure a natural benefit for every class of user on which a software system depends

➠ If there is no natural benefit, invent a lollipop

  - a software function that is not naturally part of the system functionality, but which provides enough benefit to encourage use

Systematizing the Domain

- We want to go from a Ptolmeic universe to a Copernican universe
  - A clean specification with general rules and few special cases
- The user sees epicycles, and at first so does the analyst
  - Usually there is an (almost) orderly system, but it is not easy to find
  - Strange but true: Humans can use rules without being aware of them.   Example: Language.

Rule Discovery and Test

- Similar to scientific method
  - Observe cases (procedures, special case rules)
  - Hypothesize general rule
  - Test hypothesis
    - Probably can't just ask
- Checking rule validity
  - It is difficult for ananalysts or users to understand the consequences of a rule
    - quantification ("all", "some", "never") is particularly hard
  - Examples ("experiments") can help

Examples as "Experiments"

- If a rule is valid, then all of its consequences should be valid
  - It is easier for the user to judge the validity of particular examples than of the general rule
- Try to "cover" the rule
  - Consider the "typical" case
  - Consider "boundary" cases
  - Especially consider "vacuous" cases of quantifiers
    - e.g., if rule says "if all foo are pink", consider no foo

Using Redundancy

A general technique for identifying and repairing faulty information

- Redundant examples
  - Vary factors that shouldn't matter (check for hidden variables)
- Multiple reports
  - Different users, with different viewpoints should confirm rules
    - a good confirmation must be capable of invalidating the hypothesized rule; avoid bias toward the original interpretation
  - User should re-confirm (using a few different examples) on another occasion

Scenarios

- Hypothetical situations and activities
  - a “storyboard” is a presentation of a scenario
- Help the user describe requirements through examples
- Help the user and analyst test rule consequences
  - Like experimental design in the sciences, look for consequences that could *dis*confirm a hypothesis
  - Confirmation through strange consequences is more convincing than obvious consequences

Asking questions through scenarios

- “Suppose the furnace is in normal operation, and then a wild value is recieved from the sensor. How should the furnace system react?”
- Look for general rules in the examples
- Look for exceptions to the general rules

Scenarios and Prototypes

- If a prototype is produced in the requirements phase (or in an earlier turn of the spiral), it can be used to present scenarios
  - – But mockups and “cardboard prototypes” can often be good enough for requirements clarification

Exceptional Conditions

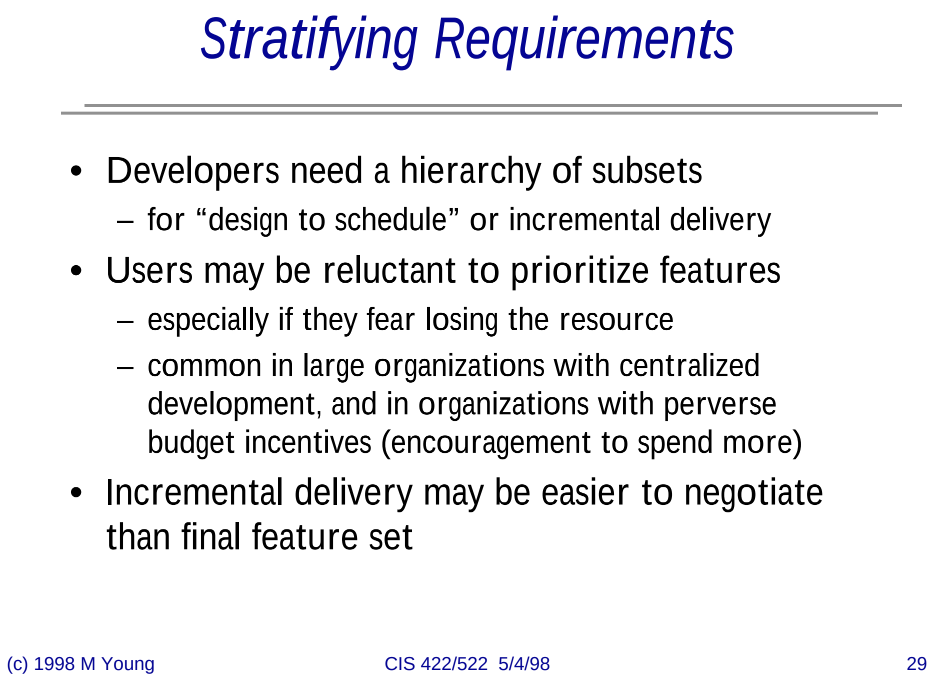
- Be careful of “always”
  - – Explicitly ask for exceptions; explore extreme cases
  - – Users sometimes say “Always X, (except when Y)”
- Some “exceptions” are really consequences of a general rule
- Some exceptions are not universally known
  - – especially: The manager may not know how the rules are *really* applied

Exploring Undesired Events

- Explore desired responses to unusual and undesired events
  - Especially when replacing a manual system. People are flexible and creative in coping with problems; software systems aren’t
- Work forward from undesired events
- Work backward from undesired outcomes
  - example: Never remove an old copy of data until a new version is in place and verified

Likelihood of Change

- For each requirement and aspect of the system, determine
  - How likely is it to change over time?
  - In what ways is it likely to change?
- Likelihood of change will guide modular organization, where we “hide” design decisions that may need to be changed
- Unfortunately, you can’t always believe what you’re told
  - Reporting of past changes is often more accurate than prediction of future changes



Stratifying Requirements

- Developers need a hierarchy of subsets
  - for “design to schedule” or incremental delivery
- Users may be reluctant to prioritize features
  - especially if they fear losing the resource
  - common in large organizations with centralized development, and in organizations with perverse budget incentives (encouragement to spend more)
- Incremental delivery may be easier to negotiate than final feature set