Software Engineering

Lecture 3 - Software Requirements

Reading:
Pragmatic Programmer Ch. 7*:* Before the Project (**MUST READ**)
S. Faulk, *Software Requirements: a Tutorial* (**MUST READ**)

Big questions

- What are requirements?
- How do we gather or find out requirements?
- How do we document requirements?
  - What to include?
  - What to omit?

Recap: Software Life Cycle

- In any software life cycle the project goes through few common steps:
  - **Requirements analysis – today**
  - Design
  - Implementation
  - Testing
  - Maintenance

Software requirements

- **requirements**: specify what to build
  - Tell "what" and not "how”
  - Tell the problem, not the (detailed) solution
- a requirement is **a statement of something that needs to be accomplished.**
- Examples:
  - An employee record may be viewed only by an authorized group of people.
  - The editor will highlight keywords, which will be selected depending on the type of file being edited.
  - Students select course offerings to register for the coming semester.

Example

The requirement is stated as:

- "Only personnel can view an employee record,"
- Here the developer may end up coding an explicit test every time the application accesses these files.

What if the requirement is stated as:

- "Only authorized users may access an employee record,"
- Here the developer will probably design and implement some kind of access control system.
- When policy changes (and it will), only the metadata for that system will need to be updated.

- "The system must let you choose a loan term“ is a req
- "We need a list box to select the loan term" - may or may not be.
- If the users absolutely must have a list box, then it is a requirement.
- If instead they are describing the ability to choose, but are using list box as an example, then it may not be.

- Work with a User to Think Like a User/Act like a user/become a user
- Input file processing is the what, parsing is the how

Why requirements?

"The hardest single part of building a software system is *deciding precisely what to build*. No other part of the conceptual work is so difficult as establishing the *detailed technical requirements*, including all the interfaces to people, to machines, and to other software systems.

Therefore the most important function that software builders do for their clients is the iterative extraction and refinement of the product requirements."

-- Fred Brooks, *The Mythical Man-Month*

Why requirements?

- 80% of the causes of failed software projects (Standish Group study):
  - Incomplete requirements – 13.1%
  - Lack of user involvement – 12.4%
  - Lack of resources – 10.6%
  - Unrealistic expectations – 9.9%
  - Lack of executive support – 9.3%
  - Changing requirements & specifications – 8.8%
  - Lack of planning – 8.1%
  - System no longer needed – 7.5%

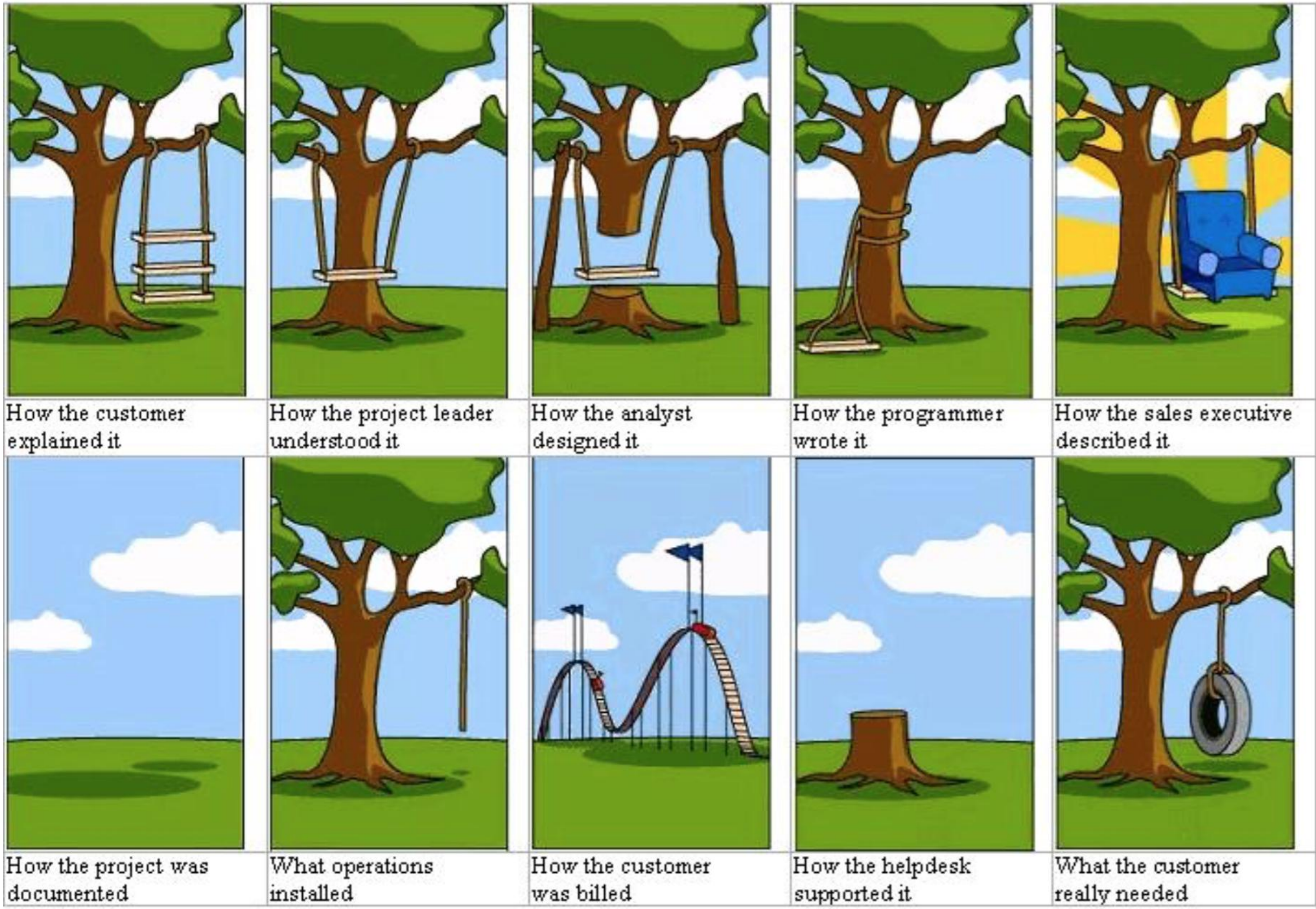
Why requirements?

- Causes of failed software projects (Standish Group study)
  - **Incomplete requirements – 13.1%**
  - **Lack of user involvement – 12.4%**
  - Lack of resources – 10.6%
  - **Unrealistic expectations – 9.9%**
  - Lack of executive support – 9.3%
  - **Changing requirements & specifications – 8.8%**
  - Lack of planning – 8.1%
  - **System no longer needed – 7.5%**

- The commonest mistake is to **build the wrong system (43%)**
- The #1 reason that projects succeed is user involvement

Why Requirements?

- They help …
  - **Understand** precisely what is required of the software
  - **Communicate** this understanding precisely to all development parties
  - **Control** production to ensure that system meets specs (including changes)


How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it
What the customer really needed

Role of requirements

- How do requirements fill the gap between various parties? Roles of requirements
  - *customers*: show what should be delivered; contractual base
  - *managers*: a scheduling / progress indicator
  - *designers*: provide a spec to design (Design is covered next)
  - *coders*: list a range of acceptable implementations / output
  - *QA / testers*: a basis for testing, validation, verification

"Digging" for requirements

How does one find out the requirements for a project?

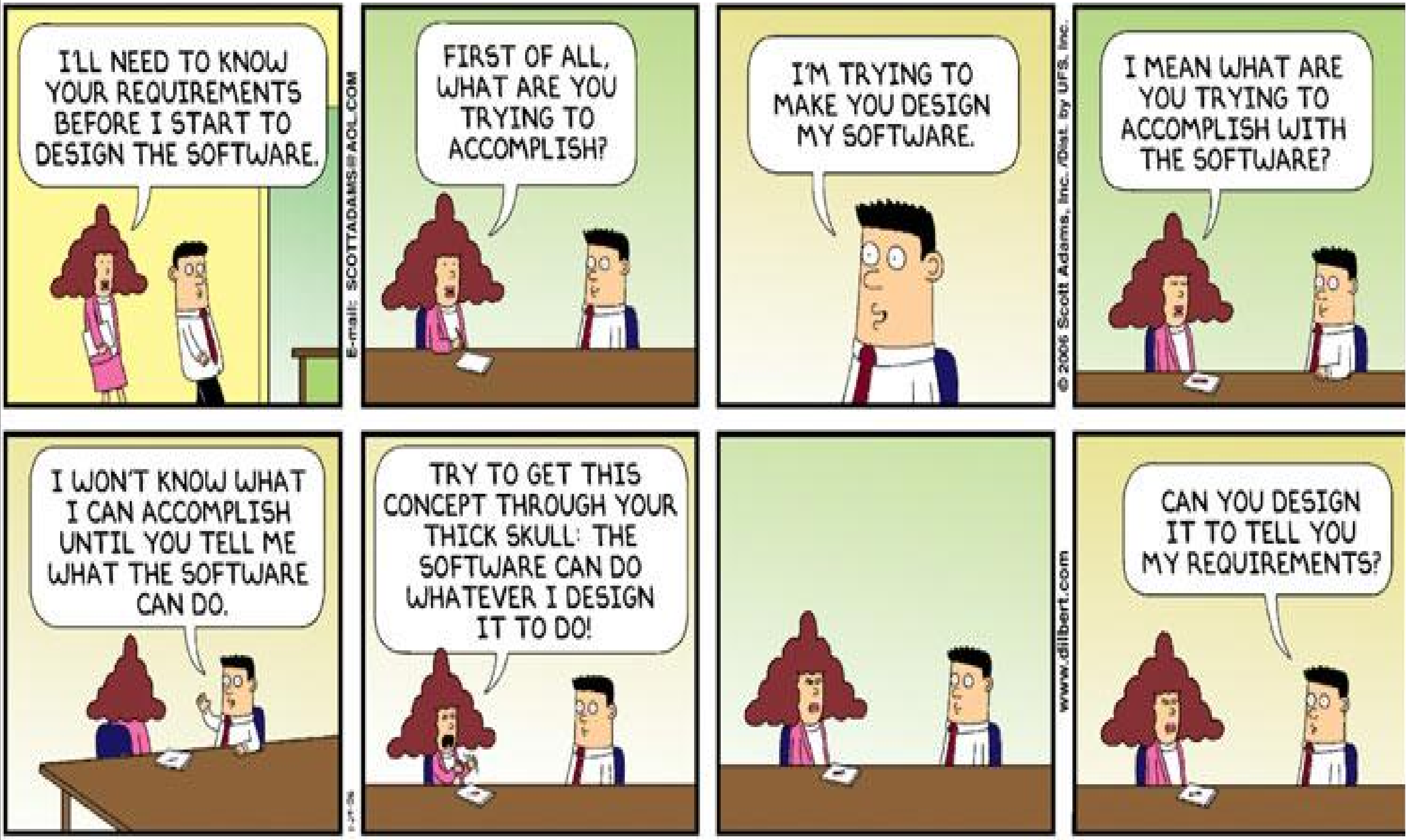
- Do:
  - Talk to the users, or work with them, to learn how they work.
    - Become the user
  - Ask questions throughout the process to "dig" for requirements.
    - Interviews
  - Think about *why* users do something in your app, not just what.
    - Understand the underlying reasons not the actions.
  - Allow (and expect) requirements to change later.
    - Continuous meetings help.
  - Write down **scenarios** of how users use your system.
  - A scenario is a tool used during **requirements analysis to describe a specific use of a proposed system**

"Digging" for requirements

How does one find out the requirements for a project?

- Don't:
  - Describe complex business logic or rules of the system.
  - Be too specific or detailed.
  - Describe the exact user interface used to implement a feature.
  - Try to think of everything ahead of time. (You will fail.)
  - Add unnecessary features not wanted by the customers.

Requirements Gathering


I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?
E-mail: SCOTTADAMS@AOL.COM
© 2006 Scott Adams, Inc./Dist. by UFS, Inc.
www.dilbert.com

Tip #1

- **policy**: A set of ideas or rules; a plan of what to do.
- **mechanism**: A process, system, or technique to get a result.
  - It's important to decouple policy (what) from mechanism (how).
  - Mechanisms should not dictate or overly constrain your policies.

- Q: How much policy information should be in requirements?
- A: Some, but not much detail. Don't include business policy.
  - *Good:* “Only authorized users may access an employee record."
  - *Bad:* "Only the administrator may view employee records."

Tip #2

- It's important to decouple requirements (what) from implementation (how).
  - *Bad* : The system must let you choose a loan term using a combo box
    - Is an implementation
  - *Good* : The system must let you choose a loan term

- Rule of tomb:
  - Don’t include implementation details in requirement unless absolutely necessary. (The client demands them)

Tip #3

- **feature creep/bloat**: Gradual accumulation of features over time. (increase in scope)
  - Often has a negative overall effect on a large software project.
- Why does feature creep happen? Why is it bad?
  - Because features are "fun"
    - developers like to code them
    - marketers like to brag about them
    - users want them
    - ... but too many means more bugs, more delays, less testing, ...
- Point out each new feature's impact on the schedule to the project team.

Tip #4

- **Implementable:** Developers should be able to imagine a realistic implementation of any requirement
  - *Bad*: The system must process stock trades between London and Chicago with nanosecond latency
  - *Bad* : The system must identify errors in user input with 100% accuracy

- **Verifiable:** There must be a way to unambiguously verify whether a requirement has been met
  - *Bad* : The system should be easy to use
  - *Better* : Typical internet users should be able to use the system after completing a short tutorial

Tip #5

- It is easy to duplicate knowledge in the specifications, or code
- when you do so, you invite a maintenance nightmare
  - one that starts well before the application ships

- Follow the ***DRY*** **principle**:
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
  - E.g changing the tax should be done in one place.

Tip #5

- Y2K (Year 2000) problem
  - In 19xx dates were represented using two numbers by businesses
    - E.g 87 represented 1987
  - Software was also designed that way – every system failed in 2000 (00 according to the computer). Why?
    - It simply figured the day was 1900
- Y2K was (in a sense) a requirements problem.
  - coders didn't consolidate date logic in one place for easy change
  - should have had a requirement such as:
    - "The system will be designed for expandability such that it can be easily modified later to work in years 2000 and beyond.”
- “Premature optimization is the root of all evil.” -- Donald Knuth(father of the analysis of algorithms)

Tip #6

- Users and domain experts will use certain terms that have specific meaning to them.
  - users and developers may refer to the same thing by different names or, even worse, refer to different things by the same name
- Create and maintain a ***project glossary***
  - one place that defines all the specific terms and vocabulary used in a project.
  - All participants in the project, from end users to support staff, should use the glossary to ensure consistency.

Good or bad?

- Which of the following are good requirements? Why/why not?

1. The system will enforce 6.5% sales tax on Addis Ababa purchases.
2. The system shall display the elapsed time for the car to make one circuit around the track within 5 seconds, in hh:mm:ss format.
3. The product will never crash. It will also be secure against hacks.
4. The server backend will be written using PHP or Ruby on Rails.
5. The system will support a large number of connections at once, and each user will not experience slowness or lag.
6. The user can choose a document type from the drop-down list.

Classifying requirements

- The *classic* way to classify requirements:
  - **functional**: high-level **functions** the system should perform, little to no reference to underlying technology issues – map inputs to outputs.
    - "The user can search either all databases or a subset."
    - "Every order gets an ID the user can save to account storage."
  - **nonfunctional**: other constraints
    - performance, dependability, reusability, safety
    - "Our deliverable documents shall conform to the XYZ standard."
    - "The system shall not disclose any personal user information."

Case Study

- Requirements for "online registration" system at AAiT
  - The new system will allow students to select five course offerings for the coming semester.
  - Each student will indicate two alternative choices in case a course offering becomes filled or canceled.
  - No course offering will have more than hundred students or fewer than three students. A course offering with fewer than three students will be canceled. (Non Functional)
  - Once the registration process is completed for a student, the registration system sends information to the billing system so the student can be billed for the semester.

Case Study

- Requirements for "online registration" system at AAiT
  - Professors will indicate which courses they will be teaching, and to see which students signed up for their course offerings.
  - Students must be able to access the system during a period of time to add or drop courses.
  - Registrar will maintain information about students, teachers, and curriculum.

Next: Documenting requirements

- Requirements are documented in a document called a *Software Requirements Specification*(SRS)
  - Next classes: (Lectures cover) Analyze and document the requirements
  - Next classes: (Assignment covers) You will create a System Requirements Specification document

  Task for next week
  - Define your systems requirements; both functional and non-functional
  - Even when your software cycle is "Evolutionary", you need to come up with a good set of initial requirements
  - **READING**: See at the beginning of the lecture

Summary

- Getting the requirements right is the single most important (and hardest) task in a large software engineering project.

- Talk to end-users but watch for feature bloat.

- Don't Repeat Yourself.

THE PROJECT REQUIREMENTS ARE FORMING IN MY MIND.
NOW THEY'RE CHANGING... CHANGING... CHANGING... CHANGING... OKAY. NO, WAIT... CHANGING... CHANGING... DONE.
NATURALLY, I WON'T BE SHARING ANY OF THESE THOUGHTS WITH ENGINEERING.
I BUDGETED FOR SOME GOONS TO BEAT IT OUT OF YOU.
S. Adams
E-Mail: SCOTTADAMS@AOL.COM
© 1994 United Feature Syndicate, Inc.