Objectives

- Discuss issues associated with functional specifications
- Identify best practices to increase your success rate

Functional specifications

- What are they?
  - Documents describing how the product should behave
- Things to include
  - Features of the product
  - Usage scenarios and user profiles
  - How to use the product screen by screen
  - Flow of action
  - Expected behavior

Why do we need them?

- Identify features
- Figure out what technology do we need
- Determine expertise needed to build the product
- Understand major components and break down into layers
- Identify risk areas and limitations
- Focus your development effort on satisfying specs
- Setup schedule based on all of the above

How do get them?

- Interview client
  - What do they want to do?
- Design patterns
  - What solutions can be used?
- Prototype features
  - How do major components operate?
  - What features a certain technology provides?
- Experience from developers
  - What things work and what won’t?

Part I: Organizational Issues

- Before taking any project and writing any code ask yourself:
  - Is my organization ready to develop software?
- Some people believe good developers is all you need
  - Reality: talent is over rated.
  - Discipline is the key to success
- Joel Spolsky – former Microsoft Excel PM
  - Internet blog with many rule of thumbs and ideas
  - Some are not right IMO
Joel Test: 12 Steps to better code

Test 1: Do you use source control?
- SVN, CVS
- Manage code and integrate with the rest
- Keep backups for free...

Test 2: Can you make a build in one step?
- Start your application top-down
  - Phase 1 of DB Project
- No mystery to compile, deploy and run application
- Most IDE create a project that runs!
- CMSC 435 @ UM - Software Engineering course
- Deliverable - software application with one click installer

Test 3: Do you make daily builds?
- Make sure you new code
  - Works and does not breaks someone else code
- ICOM 5016 last day integration syndrome
- Do it when people are around to fix it
- Rotate who is responsible for the build
  - But if someone breaks it that person should fix it

Test 4: Do you have a bug database?
- Track known bugs
  - Pick the ones to fix now and the ones to be left for future
- Track cause, buggy behavior, expected behavior, owner

Test 5: Do you fix bugs before writing new code?
- Critical bugs must be fixed ASAP
  - Ex. Null pointers, number overflows, etc.
- You know what are doing and is easier to track what happened
  - In one week you will forget what the code was doing...
- Lots of unfixed bugs = unreliable schedule to finish
  - ICOM Software Gurus
- Write 5000 lines of undebugged and untested code
- Expect to be able to fix them a week before deadline
- Often they get bored and quit the project (go to play games)

Test 6: Do you have an up-to-date schedule?
- Schedule is not carved in stone
- Each developer must update time to end task
  - Make sure debugging and testing is included
- Do not let manager change time!
  - Project will fail
- Cut luxury features in order to meet deadline

Test 7: Do you have a spec?
- Functional specification — what the software will do?
  - Not UML, not layer diagram
- Test and possible GUI sketch
  - What will happen when people use the code
- No spec = guessing

Test 8: Do programmers have quiet working conditions?
- People like to concentrate and write code (inspiration)
- Distractions
  - Phone
  - Constant questions about schedule or windows crash
  - Far away bath rooms / food / coffee
  - Co-worker interruptions

Test 9: Do you use the best tools money can buy?
- Do not torture your developers with
  - Old machines with small monitors
  - Disk space quotas
  - Outdated OS release
  - Bad software tools
    - Microsoft Paint vs. Photoshop for Web imaging

Test 10: Do you have a spec?
- Functional specification — what the software will do?
  - Not UML, not layer diagram
- Test and possible GUI sketch
  - What will happen when people use the code
- No spec = guessing

Test 11: Do you use the best tools money can buy?
- Do not torture your developers with
  - Old machines with small monitors
  - Disk space quotas
  - Outdated OS release
  - Bad software tools
    - Microsoft Paint vs. Photoshop for Web imaging

Test 12: Do you use the best tools money can buy?
- Do not torture your developers with
  - Old machines with small monitors
  - Disk space quotas
  - Outdated OS release
  - Bad software tools
    - Microsoft Paint vs. Photoshop for Web imaging
Joel Test: 12 Steps to better code

Test 10: Do you have testers?
- No: reality: Every software coding effort is full of bugs
- Yes: dedicated tester check whole system or subsystem
- Unbiased
- Tries several scenarios and documents anomalies
- Testing and coding should be interleaved
- Write code, debug, test, write code, debug, test, ...

Reality: Every software coding effort is full of bugs

Programmer does first test
- Single

Dedicated tester check whole system or subsystem
- Unbiased
- Tries several scenarios and documents anomalies
- Testing and coding should be interleaved
- Write code, debug, test, write code, debug, test, ...

Software Products classification

Products can be classified as
- Shrink wrap
  - Customized
- Throwaway
  - Targeted to a general audience
  - Ex. MS Office, Photoshop, iTunes
  - Customized
  - Specific to a given user or industry
  - Ex. CESCD David, UPR PATSI, Universal Insurance Claims Management
  - Throwaway
  - Internal code used to experiment with a given technology
  - Ex. Phase 1 and Phase 2 of ICOM 5016 Project

Shrink wrap Software

- Used by a large number of people
- Little control on how it is used
- Sell at retail stored or over the Web
- Develop and release it to the public
- Bug fix and must be provided over Web
- Scales well in terms of money
  - License issued to individual users
  - Should be able to recover cost with first N licenses
  - After that is all profit
- Need to test and maintain aggressively
  - To continue selling it and making profit
  - Create loyal customer base

Customized software

- Also called internal software
- Used by people at a company or community
  - Smaller audience
- More control on how it is used
  - You can actually dictate requirements for usage
- Develop and deploy to the company/community
  - Need to give them training
  - Often system is buggy and you need to keep fixing it
- Less scale in term of profit
  - Contract-based: Once contract is over you get no money
  - Contracts then be to expensive (to account for profits vs loses)
  - Contract expires and no more maintenance is given
  - Unless a maintenance contract gets setup

ICOM 4.0 GPA Students
- Some of them cannot write code
- They even evade ICOM 5016
- Show people you UI and collect data on
  - Problems with locations of buttons, menus, etc.
  - on

ICOM 4.0 GPA Students
- Some of them cannot write code
- They even evade ICOM 5016
- Show people you UI and collect data on
  - Problems with locations of buttons, menus, etc.
  - on
Software Products classification

- Throwaway
  - Internal code used to experiment with a given technology
  - Sometimes this is how to polish your specifications
    - Rapid prototype to figure out what you can and can't do!
  - You want to use throwaway as a means to an end
  - You do not sell throwaway software
  - Ex. Phase I and Phase II of ICOM 5016 project
    - Hardwired servlet code and in-memory DB is not use anymore
    - But you get Web-based UI and organization of beans right

Making money on software

- Shrink wrap
  - Make a product that many people will use
    - Microsoft, Apple, IBM, Adobe, Skype
  - Customized
    - Make a product that a big agency will use
    - Only do customize to help you get cash to make another product
    - Shrink wrap is where you want to be

Part II: Procedural Issues

- Software development is cyclic!
  - Old school water fall software development process assures failure
  - You need to have constant testing and feedback from the user
  - UML will not produce code for you!
    - How do I specify a multi-threaded system with a shared queue that controls access to a pool of disks?
    - UML is good to talk with others about your code
    - Like ER diagrams
    - Source code == real software specification

Cowboy Coding Model

- You start writing code without an actual plan
  - Hacker's way of doing things
    - I will start writing code and I will figure out things along the way
    - Many ICOM Software Gurus work like this
  - You guarantee that the project will be
    - Late
    - Full of hard to understand code
    - Full of incompatibilities
    - Full of unusable features
    - Featuring a hard to use UI

Waterfall Model

- Software is built in steps
  - One phase leads to the next
  - If this phase is right the next will likely be right

Waterfall Model: Problems

- In each phase you deal with a bunch of uncertainties
  - Customer changes her mind about UI
  - You drop the ball with the design
    - Mixed data model with storage logic
    - Use multi-threaded when multi-process was better
  - You realize your platform has buggy support for networking
    - Ex. PDAs
  - Change is assured when building software
    - You need a way to make mid-flight course corrections

- UML will not produce code for you!
- UML is good to talk with others about your code
- How do I specify a multi-threaded system with a shared queue that controls access to a pool of disks?
- UML is good to talk with others about your code
  - Like ER diagrams
  - Source code == real software specification
At each step you might need to revisit decisions from previous phase.

**Rapid Application Development (RAD)**
- Build incomplete but functional prototype (like a demo!)
- Debug and test major components
- Involve customer by showing prototype
  - Nail down UI
  - Prevent change of accepted features ...
- Add features/fixes into prototype until you reach release status
  - Hey, but finish the product!!!
- Examples:
  - Agile Programming
  - Extreme Programming
  - SCRUM

**Agile Programming**
- Family of techniques based upon
  - Inclusion of customer into design/development
  - Short cycle to produce working code (not all features)
  - Every few weeks a new version with a set of new features is delivered
  - Test-Driven software development
    - First make the tests, then you write code that can pass them
    - Refactor code
    - Change code based on results of debugging, testing, and user feedback
  - Produce stable release as results of continuous improvement process

**Extreme Programming**
- Based on daily practices and team values
  - Customer and business people are part of the team
  - Always deliver a new working version ASAP
  - Communicate effectively with all team members

**XP Values**
- Simplicity
  - Write code that is simple, clean and straightforward
- Communication
  - Keep direct communication between customers, developers, business people and managers
- Feedback
  - Always comment on out other code, features, and issues
  - E.g., code reviews
- Courage
  - Write the code! If you mess up just refactor
    - Avoid getting stuck in perfect implementation issues

**XP Activities**
- Simple Design
  - Start with a simple system that works
  - Add new working features
- Pair Programming
  - 2 programmers work side by side on the same machine (like Spartan kings)
  - Faster, better code plus you have redundancy
- Test-Driven Development
  - Unit test and full system tests as new features are added
- Design Improvement
  - Refactoring – fix the design as you write code
  - You only know you are wrong when you see it
**SCRUM**

- XP can be chaotic
- Scrum is controlled chaos
- The Team:
  - Scrum master
  - PM
  - Product Owner
  - Customer and business people
  - Developers
- Team works in sprints or burst of one month
  - Design, code, test and demo software
  - Next sprint adds features to previous release
  - Backlog of the spring list the features to do in each sprint

**SCRUM Process**

- 24 h
- 3-5 days
- 1-2 weeks
- 1-2 months

**Software System Architecture**

- Start out by giving high level system organization
- Boxes and arrows

```
Client Application -> Database Engine -> Disk
```

**Layered Software Design**

- Break down software model into layer
- Each layer is one or more libraries with specific role

```
Client API
Query Optimization
Query Processing
Storage Engine

Disk
```

**Each Layer is Simple**

- At this level you can lay down the classes
- UML can help you illustrate structures and relations

**Design Patterns**

- Well understood and documented recipes to build software
  - Reusable code
  - Idea borrowed from architecture
  - Archetypes
  - Columns, arcs, etc.
  - Smalltalk had them for GUI
  - Gang of Four Book (GoF) popularized design patterns for CS
  - You should build your libraries around them
Example: Abstract Factory

- You need to write an email client
- Must run in
  - Windows XP and Vista
  - MacOS X
  - Ubuntu
- Each one has a different look and feel
- You do not want to write the different programs
- Instead you want to share as much code as possible
  - Only differentiate in how UI elements are created

Questions?