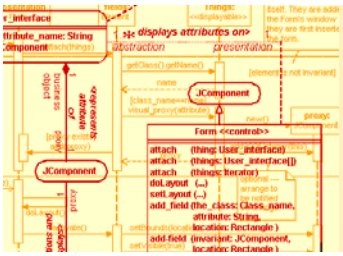


Object-Oriented Design

Allen I. Holub
Holub Associates
www.holub.com
allen@holub.com

About Me

- Working in the industry since 1979
- 9 books, hundreds of magazine articles.
- CTO, architect, programmer
  - Programmed in Assembler, Pascal, PL/1, C, C++, Java, Perl, ...
- Consult in all aspects of OO-Design
  - Adoption strategy and tactics
  - Architecture
  - Project mentoring
  - Education, not "Training" (Java and OO-Design)
- Teach for University of California, Extension

The Central Problem

72% of all software projects fail.

- Standish Group's (www.standishgroup.com) *2000 CHAOS Report*, covers the period from 1994-2000
- 40-60% of these failures are caused by lack of design and requirements gathering.
  - The software works, but it doesn't do anything useful.

OO Is Not What You Think It Is

- People think that Object Orientation means objects, classes, design patterns, UML diagrams, Java/C++, etc.
- They're wrong.
- OO is all about process.
  - Good design cannot happen outside the context of good process.
  - Good programming cannot happen outside the context of good design.
- We're going to cover the whole process.
  - If that upsets you, you may as well leave now.

 Allen Holub's OO-Design Workshop www.holub.com 4

Why Do It? (1)

- OO code is much easier to build and debug than procedural code.
- Typically, the cost of moving an existing system to OO, no matter how large the system, is recouped the first time a significant maintenance issue comes up.

 Allen Holub's OO-Design Workshop www.holub.com 5

Why Do It? (2)

- OO Systems solve the real problems of real users.
  - OO organization mandates that the program reflects the actual business model in the user's mind.
- Every hour spent in design replaces 3 hours of coding/debugging.
- Helps you keep your best employees.
  - Once you "get" it, it's actually fun.
  - You can be extremely productive in a 40-hour work week

 Allen Holub's OO-Design Workshop www.holub.com 6

Why not do It? (1)

- Moving to OO typically requires extensive training of both existing and new staff.
  - management training is essential and often omitted.
- OO is really a new way of thinking, difficult to pick up from a book, so requires lots of interaction with experts to learn,
- It's rare for a programmer off the street to understand OO even superficially.

Why not do it? (2)

- Disruption
  - The core structure of the software oganization might have to change
- Some programmers resist the change so much that there will be no place for them in the new organization.
  - Amongst this group will be the grizzled gurus who hold the entire program in their heads, and are unable (or unwilling) to write English rather than code.

Myths

- OO==A radical shift in thinking
- OO==good
- OO==new (dates from 1967.)
- OO takes longer
- You need certain tools (Rose, Java) to do OO.

Realities

- OO *requires* a radical shift in thinking.
- OO systems are faster to build and are easier to maintain.
  - Bugs in OO systems are localized.
- OO systems are very flexible, easily adaptable to new business requirements.
- OO systems do what the users want.
- OO processes are for grown ups.
  - Fire the "cowboys."
- OO is understood by perhaps 5% of the programmer population.
  - Most claimed OO systems are not in the least bit OO.

 Allen Holub's OO-Design Workshop www.holub.com 10

Determinism and Complexity

- *All* large computer programs are nondeterministic,
  - Given an arbitrary set of inputs, it's impossible to predict the outputs with 100% accuracy.
- So don't act as if they were.
  - Program assuming that nothing works (now or ever) and that everything *will* change.
- OO renders complexity manageable, but doesn't eliminate it.
  - OO systems tend to be larger, more complex, and vastly easier to maintain than equivalent procedural systems.

 Allen Holub's OO-Design Workshop www.holub.com 11

Conway's Law

The structure of a system tends to mirror the structure of the group producing it.

—Mel Conway (April, 1968 Datamation)

- Introducing a new process requires you to change the structure of the organization.
- These changes are often both radical and disruptive, but are often fun and interesting as well.
- If things were working great the way they are, you wouldn't be looking at alternative methodology, would you?

 Allen Holub's OO-Design Workshop www.holub.com 12

Resistance

- Institutional (turf wars)
  - The Directory of QA won't be happy when the QA department is dispersed amongst the design teams
- Personal
  - OO isn't "right"
  - I won't be an expert any more.

Correcting Some Common Misconceptions

- Design is about communication, not coding.
- Collaboration ≠ negotiation
- Documentation ≠ Understanding
- Discipline ≠ Formality
- Process ≠ Skill
- Experience/Knowledge ≠ Effectiveness

Good Designs Provide the Minimal (Least Costly) Solution.

- Do exactly what's required. Nothing more. Nothing less.
- Solve the user's problem. Period.
- But design the system to be flexible enough to accept change.

Written Communication Is Ineffective

- In order of effectiveness: (Cochburn)
  1. Two people at a white board
  2. Phone conversation
  3. Email
  4. Video Tape of someone at a whiteboard
  5. Audio Tape of someone at a whiteboard
  6. Paper

Written communication is ineffective… BUT

- Written artifacts of the design process are essential parts of the process itself.
- You cannot design (period) unless you can transcribe your thoughts to paper with precision.
  - Which came first, language or thought?

Beware the cargo cults

- Process ≠ Artifacts
- Don't confuse the artifacts (UML, Requirements documents, etc.) with the process itself.
- Simply following a process does not guarantee good results.
- Simply creating a set of documents does not mean that you'll create a good program.
- There are such things as bad designs.
- See Steve McConnell's Editorial:

  http://computer.org/software/so2000/pdf/s2011.pdf

Artifacts Are Unimportant

- The point of design is to think deeply about the problem, not to produce documents.
- The various artifacts of design (UML, etc.):
  - make your thinking concrete.
  - help you present that thinking to others and work collaboratively.
  - provide documentation for the finished product.
- It's the act of creating the documents that's important, not the documents themselves.
  - Writing things down clarifies your thinking.
- It doesn't matter if a document changes.
  - Changes just mean that you've learned more about the problem.

Brook's Law

"Adding manpower to a late software project makes it later."

- In general, larger groups are less productive.
  - The larger the group, the more difficult the communication.
  - There are exceptions when simultaneous parallel activities can go on:
    - testing and coding.
    - Drawing on a whiteboard, using a CAD program/CASE tool.
- See: Fred Brooks, *The Mythical Man Month, Anniversary Edition* (Reading: Addison Wesley, 1995).

No Silver Bullets

- You spend 70% of your time thinking (more if you don't design).
- The best improvement an automated process can give you is 30% (unless you change the way you think).
- E.g. Longs & Forté ($30-$40Million)

Fred Brooks, *The Mythical Man Month, Anniversary Edition* (Reading: Addison Wesley, 1995).

Programming is Writing

- Properly written, grammatically correct, fully-formed English is essential.
- "It's trivial" means "I don't understand the problem fully and can't be bothered to think about it now."
- If you can't say it in English, you certainly can't say it in Java.
- The OO process has more to do with formal linguistic modeling than with traditional programming.

Tools & Environment

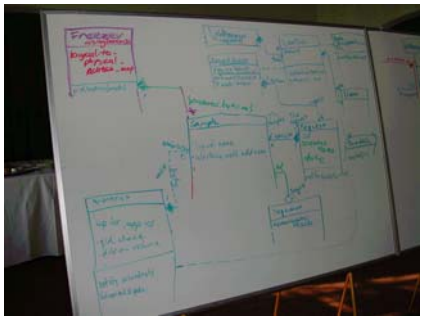
CASE Tools

- For the most part, not worth the money.
- Round-trip engineering doesn't work.
  - Design, code, reverse-engineer ...
  - Encourages ad-hoc programming with no design work up front.
- None of them integrate the entire design process.
  - Rational Rose is particularly worthless.
- But if you insist:
  - MagicDraw (www.magicdraw.com) Java drawing program. Quirky. Incomplete.
  - Together (www.togethersoft.com) has good graphics.
  - ArgoUML (www.argouml.org) is free, written in Java, and just as good as the others.

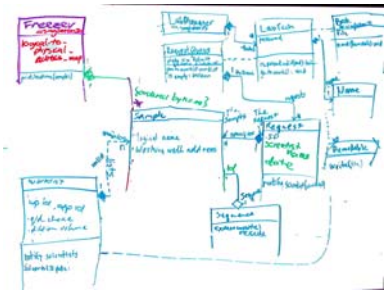
Tools I Use

- Bound engineering notebooks.
  - U.S. patent law awards the patent to whoever came up with the idea first. Filing date is irrelevant.
- Sketch pads.
- Giant post-it pads.
- White boards.
- A digital camera & Whiteboard Photo
- ArgoUML if pressed (www.argouml.org)
- A drafting program (Visio, AutoCad) only if client demands pretty pictures.

Whiteboard Photo (before)



Whiteboard Photo (after)



- http://www.websterboards.com/products/flash.html

The Environment (1)

- A comfortable *quiet* dedicated design room is essential:
  - Conference rooms don't work
  - No telephones.
  - Couches.
  - LCD Projector for written docs.
  - *LOTS* of white-board.
    - Movable ones are great
    - Put the boards where the people are. (Halls, break rooms…)

The Environment (2)

Gerald McCue, "IBM's Santa Teresa Laboratory---Architectural Design for Program Development" (*IBM Systems Journal* 17:1, pp. 320-341) get from www.holub.com/goodies/.

Tom DeMarco & Timothy Lister, *Peopleware: Productive Projects and Teams.*

- 100 sq. ft./programmer, 30 sq. ft. of desktop.
- Sssssh!
  - A 1-second distraction costs 15-minutes of productivity.
  - Are people hiding in conference rooms trying to get work done?
  - Mute the phones, disable the PA system.
- Natural light (windows) increase productivity
- 4 people per office (all members of the same team).
- A door that closes.

The Environment (3)

- NO CUBES.
- A village: ideal workspaces grow organically, and are individualized by their residents.
- Irregular shapes are desirable
- Lots of whiteboards, scattered all around.
- Work gets done during casual conversations (in the hall, in the kitchen making coffee).
- Put whiteboards everywhere people congregate.
- An outside garden.

The Room (XP version)

- Dedicated Workstation clusters, not assigned to individual programmers.
- 2 programmers per workstation!
- Whiteboards around the perimeter.
- Integration system on the side.

Methodology and Process

Management Books

- Adele Goldberg and Kenneth Rubin, *Succeeding with Objects: Decision Frameworks for Project Management* (Reading: Addison Wesley, 1991).
- Tom Demarco and Timothy Lister *Peopleware : Productive Projects and Teams, 2nd Ed.* (New York, Dorset House, 1999).
- Tom Demarco *The Deadline : A Novel About Project Management* (New York, Dorset House, 1997).

More Management Books

- Gerald Weinberg, *The Psychology of Computer Programming, Silver Anniversary Edition* (New York, Dorset House, 1998).
- Alistair Cockburn, *Surviving Object-Oriented Projects: A Manager's Guide* (Reading: Addison Wesley, 1998).
- Steve McConnell, *Rapid Development* (ISBN:1556159005)

Methodology is like Elephant Repellent

"What's that?"
"Elephant repellent, I just got it for 100 bucks"
"100 bucks, are you crazy? There are no elephants around here!"
"See, it works!"

Beware the Methodologists

- In the pursuit of good software, methodologists usually assume that it was *their* methodology that brought about an improvement.
- They overestimate their own importance.
  - Everyone works better when a high-priced consultant looks over their shoulder.

"Software Engineering" Is An Oxymoron

- Building software is a cultural activity.
- Booch equates building software to playing a game.
- A dance, not a sport. (Participants cooperate, not compete.)
- The artifacts of the design process are all primarily communication aids.
- For example: UML is semantic modeling.
  - You're doing linguistics, not math.
- Large projects require more process because communication is more dificult.

Good Software Is Created by Good People.

- OO requires excellent people.
  - both design and programming.
- *All* methodologies require excellent people.
  - DeMarco: "Superbly Trained" people
    (*IEEE Computer* 35:6, p. 90)
- 10:1 rule:
  - A program that can be put together by 10 excellent programmers will require 100 average programmers.

They're Not Lying; They're Clueless.

- Many people use OO technologies procedurally.
  - They think they know OO, but they're wrong.
  - They'll (honestly) claim OO ability on a résumé.
- Experience with specific technology isn't meaningful.
  - Java, C++, EJB, J2EE, AWT/Swing, Etc., can all be used procedurally.
- Good designers have strong:
  - verbal and written communication skills.
  - "people" skills.
  - organizational skills and self discipline.
  - an ability to think abstractly
  - technical abilities

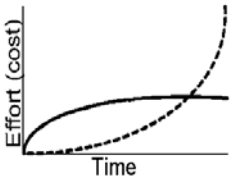
Superb "Training"

- It's best to hire smart people, then educate them.
  - "You *educate* people. You *train* animals."
- "Superb" training requires
  - Education
    - You can't learn OO solely from classes and books.
    - You must learn from somebody who's done it.
  - Mentoring
    - You have to do it to "get" it.
    - You won't get it right on your own.
  - Time
    - Good design is as hard to learn as programming.
- Hire a consultant to help.

Good Process is Organic

- Externally imposed processes are usually rejected within a few months.
- Teams must define their own methodologies.
- Select processes that make sense for your culture.
- Continuously refine your processes.
  - Toss the ones that don't work.
- Document your processes.
  - change the document as necessary.
- Based on "best practices" with proven track record.
  - Steve McConnell, *Rapid Development* (ISBN:1556159005)

The Ideal Development Curve



- The solid line is the (achievable) ideal.
- The dashed line is what we're afraid of.
- The only way to flatten the curve is through good design.

OOD is a Formal Process

- Cannot do OO in an ad-hoc way, even for trivial programs.
- Up-front design is critical.
  - OO systems are more complex than procedural ones, but manage the complexity more effectively.
- Good process does not mean that you'll have a good program.
  - There's such a thing as a *bad* design.

"Process" Means Many Things

- Heavyweight processes work well when the entire problem can be full defined before development begins.
  - E.g.: SEI, RUP
- Lightweight processes work well when the problem definition changes as development is in progress.
  - E.g.: Extreme Programming (XP), Feature Based design.
- Agile processes are essential when requirements change.
  - Agile ≠light.

OOD Is Done In Teams

- Like Volvo builds cars.
- Brooks/IBM "Surgical-Team" Model
  - Chief Architect (surgeon)
  - Copilot
  - Toolsmith
  - Tester (1:1 programmer:tester min.)
  - Domain Expert (A real end User)
  - Language Lawyer (English and Java)
  - Clerical/Drafting support
  - Secretary/Document Manger

Brooks/IBM: "Producer/Director"

- The director makes the movie.
- The producer makes sure the director can work.
- Must be equals:
  - Same salary.
  - Same perks.
- Can't work in a non-collaborative environment.

Producer/Director (2)

- Product Manager: works for marketing. Owns the specification and feature set.
- Lead Programmer: works for engineering. Owns the schedule.
- Project Administrator: works for operations, facilitates. Owns the budget.
  - Does the laundry, walks the dog, buys the equipment.

Essential Process Elements

- Think before you code (design)
- Regular code/design inspection
- Adherence to coding standards
- No ownership of pieces of the program
- Automated regression testing
- Incremental development (short cycle times)
- Document management
- Source-code control

SEI Capability Maturity Model

1. Initial chaotic.
2. Repeatable policies for managing software are established.
3. Defined standard processes in place across the organization.
4. Managed quantitative goals are set and met.
5. Optimizing focus on process improvement.

You must be operating at SEI level 3 to full leverage OO design/development.

SEI CMM Is Proven

- CMM is Scalable (PSP/TSP)
  - Watts Humphrey, *Introduction to the Personal Software Process* (ISBN: 0-201-54809-7).
  - Watts Humphrey, *Introduction to the Team Software Process* (ISBN: 0-201-47719-X).
- Can be used to formalize "agile" processes:

  "When rationally implemented in an appropriate environment, agile methodologies address many SW-CMM Level 2 and 3 practices. The ideas in the agile movement should be carefully considered for adoption where appropriate..." *(Mark Paulk, "Agile Methodologies and Process dicipline," CrossTalk, Oct. 2002, pp.15-18).*

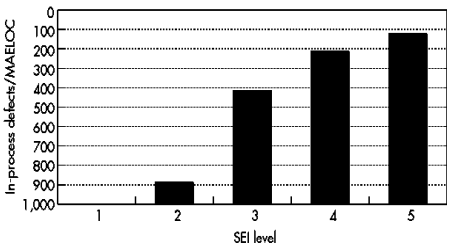
Putting CMM Into Perspective

- Works well with very large, high-ceremony, low flexibility projects.
  - 318 "best practices" organized into 18 "key" process areas followed rigorously.
  - Mark Paulk et al, *The Capability Maturity Model: Guidelines for Improving the Software Process* (ISBN 0-201-54664-7)
- Programmers are "fungible assets."
- Can't handle rapidly-changing requirements.
- Guarantees repeatability, maintainability, on-time delivery, low defects.
- Does not guarantee that the software does anything useful.

So, What Good is SEI?

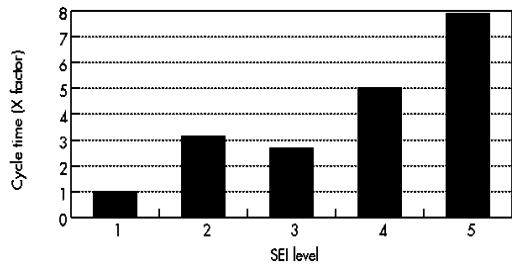
- The real value is in the activities, not the ceremony:
  - Design before you build (Level 2).
  - Standardize tools and languages across the organization (Level 3).
  - Measure what you do, and make decisions based on these measurements (Level 4).
  - Continually refine the processes, throw out what doesn't work (Level 5).
- Large projects require more ceremony.

Motorola's Experience with SEI (Defects)

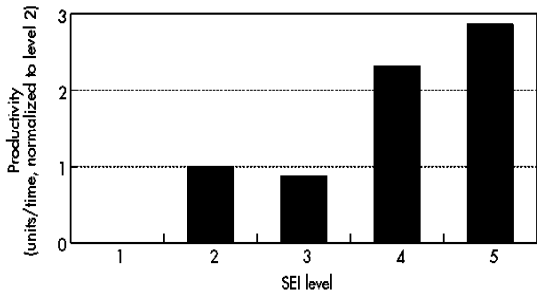


***IEEE Software* 14:5 (September/October 1997), pp. 75-81.**

Motorola's Experience with SEI (Development Time)



Motorola's Experience with SEI (Overall Productivity)



 Allen Holub's OO-Design Workshop www.holub.com 55

The "Unified" Process

 Allen Holub's OO-Design Workshop www.holub.com 56

"RUP" isn't Rational

- Call it the "Unified Process"
  - Renamed to the "Rational Unified Process" by Rational Software's marketing department.
- Grady Booch, Ivar Jacobson, and James Rumbaugh amalgamated several software-development processes then in common use.
  - They cataloged the processes, not invented them.
  - The processes described in "RUP" were not developed by Rational Software.
- No Rational-Software products are required.
  - CVS (free) is great for collaborative source-code control.
  - Innumerable tools work better than Rose for UML diagrams, and many of these are free as well.
  - Etc.

 Allen Holub's OO-Design Workshop www.holub.com 57

The Unified Process is Big

- 4 "phases," 9 core "workflows," 31 "workers," 103 "artifacts," 136 "activites," guidelines, ...
  - Mix and match from the list to fit the situation.
  - Some combination of these describe most known software methodologies.
    - Thus all software methodologies are UP!
- UP is, nonetheless, more flexible and less formal than the SEI/CMM processes.

UP Best Practices

- Develop Iteratively
- Manage Requirements
- Use Component Architectures
- Model Visually
- Verify Quality
- Control Changes

The Essentials

- It's not so much "unified," as it is a catalog of techniques from various processes.
  - When two notations/processes were in conflict, both were incorporated.
    - E.g. UML Sequence and Collaboration diagrams represent identical information in different ways. One came from Rumbaugh's OMT, the other from the Booth notation.
- No project applies all of the Unified Process.
  - Pick the part that works for you.
- We will examine a subset process suitable for small-to-medium sized teams.
  - 1-to-40 programmers

UP: Vision

- High-level description of what the system will do, what problems it solves, how it will solve them.
- Makes a business case for the software.
- Provides a check on "gold plating."

The Artifacts of "Vision"

- Glossary (key terms) ✓
- Problem Statement ✓
- Stakeholders/Users and their needs ✓
- Product Features ?
- Use cases (OO version of "functional requirements") ✓
- Non-functional requirements ?
- Design constraints ✓

"Plan the Flight, Fly the Plan"

- Project Organization, Team Structure
- Schedule
  - Project plan, Iteration plan, resources, tools
- Requirements-Management Plan
- Configuration-Management Plan
- Problem-resolution Plan
- QA Plan
- Test Plan
  - Formal test cases
- Evaluation Plan
- Product-Acceptance Plan

Risk Management

- Identify highest-risk activities early in the project and address these risks first.
- List them out on paper, then come up with a way to address them.
  - E.g. The risk that user's will reject the program because it imposes an unwieldy process can be mitigated by developing the activity flow collaboratively with the users.

Issue Management

- Formal "issue" management necessary only in very-large projects.
  - Issues include new risks and how to address them, management issues, technical issues.
- Issues are tracked like bugs
  - They're assigned numbers, and must be "retired" within a certain time frame.
  - Regularly updated at "status" meetings.

Business Case

- Programmers usually don't "get" the business case for a program, so they loose track of what's important and what isn't.
- Detailed economic plan, covers many eventualities and scenarios.

Architecture

- An overall design is essential for success, though the size and detail level of design varies with the complexity of the project.
- I'll discuss the artifacts of the design process (UML, etc.) in depth, below.

 Allen Holub's OO-Design Workshop www.holub.com 67

Component Architecture

- The UP folks believe that systems can be built using course-grained components.
- The premise is arguable.
  - The work that goes into creating a robust generic component is considerable, and the risk is high—there's no way to predict if that component will be reused, or whether it's capabilities match the requirements of the next system.

 Allen Holub's OO-Design Workshop www.holub.com 68

Configuration & Change Management

- Changes will be requested as soon as code is released
  - Almost immediately if you're doing things "right."
- Recording requests, either for new capabilities or fixes for broken ones.
  - Not "features."
  - You must prioritize requests.
  - You must track their development.
- Software version control

 Allen Holub's OO-Design Workshop www.holub.com 69

Support is Part of the Process

- Manuals, etc.
  - Should exist before the product is built.
- Reporting
  - Must feed back to design/development group.
  - It's okay for level-one support to handle RTFM problems, but if the answer is not in the manual, the question must go to a real engineer.
    - Companies that make it too difficult to get past level-one support are guaranteeing failure by keeping essential information from the developers.
  - Three categories of problems need to be tracked
    - X blows up the program (implementation)
    - I need to do X, but it's difficult or frustrating (design)
    - I need to do X, but it's impossible. (analysis)

Agile Processes

Agile Development (1)

- A set of principles, not a methodology.
- http://www.agilealliance.org/ (Alistair Cockburn, "Crystal")
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
  - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
  - Working software is the primary measure of progress.

Agile Development (2)

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Business people and developers work together daily throughout the project.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Agile Development (3)

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
  - 40-hour weeks, mandatory vacations, etc.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
  - It's hard to make things simple.

Agile Development (4)

- Build projects around motivated individuals.
  - Give them the environment and support they need, and trust them to get the job done.
  - No room for second-string people.
- The best architectures, requirements, and designs emerge from self-organizing teams.
  - Process and team structure is not imposed from outside.
  - Developers are trusted to do their work.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
  - "Postmortem" (Highsmith: Postpartum).

Extreme Programming (XP)

- An example of both Agile and "lightweight" UP.
  - Grady Booch, *Using the RUP for Small Projects* (http://www.rational.com/media/products/rup/tp183.pdf)
- Goes directly from "use cases" to code
  - Design is informal and incremental (design as you code).
- Can easily accommodate changing requirements.
- Lots of good ideas that can be applied to more rigorous processes.
  - Kent Beck, *Extreme Programming Explained: Embrace Change* (ISBN: 0201616416).
  - IEEE Software 20:3 (May/June 2003), entire issue.
  - http://www.extremeprogramming.org
  - http://www-106.ibm.com/developerworks/java/library/j-xp/

The Dark Underbelly of XP

- Often used as an excuse to abandon process entirely.
- Doesn't scale well to large groups or large programs.
- Fails miserably with undisciplined programmers.
  - XP is a formal process built on interrelated best practices.
- 80/20 rule:
  - Adopting only 80% of XP yields only 20% of the benefit.
- Provides few design artifacts.
  - Long-term maintenance is difficult.
  - The only documentation is the code.
- Pete McBreen, *Questioning Extreme Programming* (ISBN: 0-201-84457-5).

XP Options Pricing for Software

- How much money/time will it cost to add it now?
- How much money/time will it cost to add it later?
- What's the probability of it actually being used?
  - Is this SWAG (stupid wild-ass guessing)?
- Building a feature that's not used is a waste of time and money.
  - Build exactly what's required; no more; no less.
  - But build it in such a way that it can evolve easily

The System Metaphor

- A high-level metaphor that describes what the program does.
- The user's image of the system
  - should be reflected in the underlying structure.
- A good start for a design, but usually inadequate (IMHO).

 Allen Holub's OO-Design Workshop www.holub.com 79

The Planning Game: Programming to Use Cases

- The "Planning Game."
  - *Use cases* (Beck "stories") identify tasks that must be performed by a user to solve a particular problem.
  - Plan development based on use-case dependencies.
  - Develop your program one use case at a time.
    - Use cases *factored* into "activities" that can be implemented on a two-week cycle by one "pair."
  - Daily "stand-up" meetings.

 Allen Holub's OO-Design Workshop www.holub.com 80

Write the tests first.

- Start by designing the messaging system and interfaces.
- Write code in terms of the interfaces to see if the interfaces work.
  - If it doesn't, the design has failed the test.
  - E.g. Write code that uses the library before you write the library (to test if the interface to the library is usable).
- Write "stubs" that allow your test to to compile.
- Fill out the stubs one at a time.
- Test constantly.
  - Every 10 minutes.
  - After every change.

 Allen Holub's OO-Design Workshop www.holub.com 81

Constant Refactoring

- Refactoring: Modify a system to make it better without changing external behavior.
- Every time you look at a design or code, ask "how can I improve it?"
  - When you see a problem, fix it. NOW!
- You cannot refactor safely (or efficiently) without automated regression tests.
  - If the tests that worked before the change still work, you're okay.
- XP: design as you code; design changes over time.
  - You must refactor to accommodate the new design.
- Reduces code size.
  - XP projects typically ½ the size of non-XP projects with similar functionality.

Simplicity

- Design and build exactly what's required. No more. No less.
  - But do it in a way that permits easy additions.
- Use the simplest (most "transparent") algorithm.
- Simplicity requires more work than complexity.
  - The first attempt is always too complicated.
  - As the design/code improves, it gets simpler.
- Simpler code is easier to maintain and extend.
- Metrics based on quantity (e.g. lines of code per day) encourage bad design and unmentionable code.

Pair Programming & Collective Ownership

- Two programmers share a single workstation.
  - One types.
  - One worries.
- Continuous code review lowers the bug count.
- Encourages *collective code ownership:*
  - Everybody works on all the code.
  - You can go on vacation without the house falling down.
  - No surprises.
- To read further:
  - Laurie Williams et al, "Strengthening the Case for Pair Programming," *IEEE Software*, July/Aug., 2000, pp. 19–25.
  - Laurie Williams and Robert Kessler. *Pair Programming Illuminated* (ISBN 0-201-74576-3).

Code to a Standard

- Code must be self documenting:
  - transparent structure.
  - well-chosen names.
  - consistent style.
- Everyone works on all the code.
  - Code becomes unreadable without a consistent standard.
- Standard must be developed collaboratively, but someone must be "in charge" so that silly disputes can be resolved.
  - There is such a thing as a bad standard.
- Many standards are arbitrary. Live with it.
  - If you can't follow the standard, you're fired.

On-Site Customer

- An on-site domain expert, ideally an actual customer or two, is essential.
  - Refines requirements on an ongoing basis.
  - Improves efficiency.
  - Minimizes wasted effort.
  - Provides instant feedback and advice.
  - Improves the odds that the software will be useful to someone.
- E.g. Fireman's Fund: 12 best insurance adjusters reassigned to IT for a year.
- A "customer" can be a group of people.

The 35-40 Hour Week

- Kent Beck:
  - "Overtime is a symptom of a serious problem on the project...If you come in Monday and say 'To meet our goals, we'll have to work late again,' then you already have a problem that can't be solved by working more hours."
  - "No one can put in 60 hours a week for many weeks and still be fresh and creative and careful and confident."

Get it to the User's Quickly

- Two-week release cycle.
- Continuous integration.
  - New capabilities added to the code daily.
- Gets code into the user's hands quickly, so potential design problems are identified early.
- Uncovers unanticipated requirements.

Detailing A Workable Process

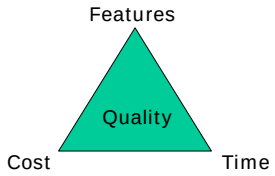
Analysis Paralysis

- Two main causes:
  - Indecision.
    - It's not clear what to do, so you discuss endless possibilities.
  - Lack of information.
    - It's not clear what the user wants, so you discuss endless possibilities.
- On-site customers eliminate both problems by providing answers.

Delivery Trade-Offs

Excessive or irrational schedules are probably the single most destructive influence in all of software.

–Caspers Jones



Estimation

- An estimate is a tool, not a bludgeon.
- Irrational estimates increase development time.
- Estimates are not negotiable.
  - but feature sets and cost (which affect time) are.
- Use Probabilities:
  - We have a 50% chance of delivering by July.
- A Spreadsheet is all you need:

Task	Estimate	Actual	Velocity
Task A	1.5 days	1.6 days	0.0
Task B	3.0 days	6.0 days	2.0
Task C	2.0 days	1.0 days	0.5
...	...	...	...

  - Multiply similar (future) tasks by velocity

Mock-ups and Prototypes

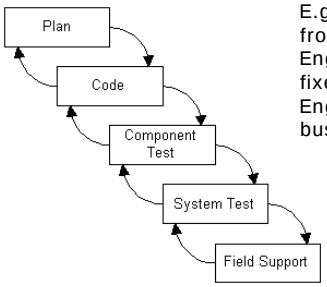
- A *Mock-up* is an informal throw-away model of some piece of the program, meant to answer a question that comes up in design.
  - E.g. "is 8-point type readable at this resolution?"
- A *Prototype* is a partially constructed program.
  - E.g. Do I have enough bandwidth for X?
- A program is a continually defined prototype.

Executable Release

- Our goal is to get working code into the hands of the users as quickly as possible.
- An *executable release* is a fully functional subset of the entire system.
  - OO systems are heavily modularized.
  - Systems built by "use case" can be made useful very quickly.
  - Executable releases are real, production code, not throwaway hacks.

 Allen Holub's OO-Design Workshop www.holub.com 94

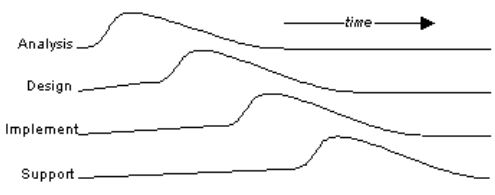
The Waterfall Model Doesn't Work (Never Did Work)



E.g.: Bugs passed from Test to Engineering aren't fixed because Engineering is too busy with version 2.

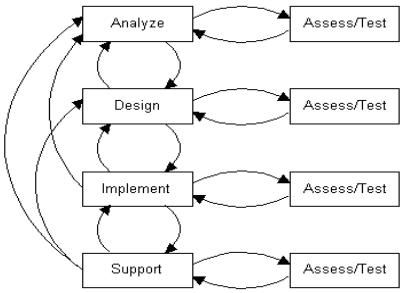
 Allen Holub's OO-Design Workshop www.holub.com 95

Four Primary Tasks



 Allen Holub's OO-Design Workshop www.holub.com 96

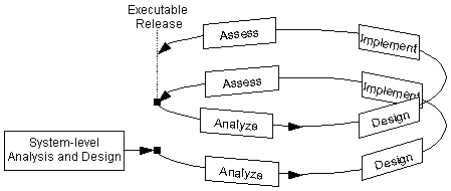
Recursive/Parallel Development



- Pervasive testing.
- Every step finds flaws in the previous step.
- Constant refactoring.

 Allen Holub's OO-Design Workshop www.holub.com 97

Spiral Development



- Two weeks around the spiral (one "cycle").
- "Executable release" to user at end of every cycle.
- Requirements will change with releases.
- Constant refactoring.

 Allen Holub's OO-Design Workshop www.holub.com 98

1. Learn the Problem Domain

- "Problem Domain:" The area of expertise in which a problem is specified.
  - E.g.: The problem domain for an accounts-payable package is "accounting."
- When designing, you must stay in the problem domain for as long as possible.

 Allen Holub's OO-Design Workshop www.holub.com 99

Communicate

- You must be able to have a conversation with a domain expert at the level of an "intelligent layman."
  - If you're doing an accounting application, read an "Accounting 101" text or take a junior-college class in accounting.

Avoid The "Bathroom Effect"

- Suppose we built houses the same way we build software?
- Puzzled owner: "Where are the bathrooms"
  Contractor: "Whadaya mean 'bathrooms,' there are no bathrooms in the spec.!"
  Angry owner: "Who'd be dumb enough to build a house without bathrooms? It never occurred to me to require them."
- Sometimes, it's the most obvious (to the user) functionality that's not mentioned when gathering requirements.
  - You need to know the domain to find these.

2. Identify User Goals

- What is the user *really* trying to accomplish?
- Knowing the goals dramatically effects the entire program.
  - BART — emerge from the system
  - Meeting Scheduler — not to go

Alan Cooper, *About Face: The Essentials of User Interface Design* (Foster City, IDG Books, 1995).

Joel Spolsky, *User Interface Design for Programmers* (Berkeley: Apress, 2001)

Conflicting Goals are Normal

- Different users often have conflicting goals.
- E.g.:
  - To teach kids to save money (Parent)
  - To show how compound interest works (Parent)
  - To keep an accurate accounting of money earned (Parent).
  - To maximize my account balance (Kid).

 Allen Holub's OO-Design Workshop www.holub.com 103

3. Working With An End User, Develop The ***Problem Statement***

- A description of the problem that needs to be solved and any domain-level solutions.
- A problem statement is <u>not</u> a description of a computer program, it is a description of the problem itself.
- An English essay describing <u>what</u> the program must do.

 Allen Holub's OO-Design Workshop www.holub.com 104

Stay in the Problem Domain

- It is essential that the problem statement be written in the vocabulary of the problem domain.
  - For example, the problem statement for an accounts-payable package should be defined in the vocabulary of an accountant, not that of a computer programmer. If there's any jargon at all, it should be accounting jargon.
  - This means that, at minimum, you'll have to read a book on accounting before you're qualified to write an accounting-related problem statement.

 Allen Holub's OO-Design Workshop www.holub.com 105

Your Audience is the Domain Expert

- A well-crafted problem statement does not mention computers or the behavior of a specific computer program.
  - Since the problem domain is "accounting," not "computer programming," Describe the problem, not the solution.

An Example Problem Statement

The Bank-of-Allen(TM)

One of the best ways to teach kids how to manage money (and how interest works) is by having a bank account. Real banks, however, don't pay enough interest to catch a kid's interest, so to speak. (At a nominal annual rate of 3.5%, $20.00 earns a big $0.72 after a year—not very impressive). Taking a cue from a piece I heard on National Public Radio's "Marketplace," I decided to open the *Bank of Allen*™ (or BofA), which pays out an effective 5%/month (that's right, per month—60% annually), but compounded daily. At this rate, $20.00 deposited in the *Bank of Allen*™ earns $15.91 over a year. The *Bank of Allen*™ otherwise works like a real bank. Kids have their own accounts, over which they have control of everything but the interest rate. They can look at (or print) their passbooks whenever they want. They can make deposits and withdrawals...

4. Identify the Use Cases

- A *use case* is a stand-alone task that has a useful outcome.
  - Logging on is not a use case.
- The complete set of use cases identify all tasks necessary to solve all problems defined in the "Problem Statement."
  - Typically, the "boundary" or analysis-level use cases are the ones that the user thinks about.
  - Other use cases define tasks that are performed internally by the program to implement the boundary cases.

Use-Case Factoring

- Large use cases can be "factored" into smaller ones.
  - "Authenticate" might be a "subcase" of several uses cases, or appear several times in the same use case.
- Use cases should be factored into pieces that take no longer than a couple weeks to implement.

 Allen Holub's OO-Design Workshop www.holub.com 109

The Essential Parts of a Use Case

- Executive Summary
  - What task is being performed.
- Detailed Description
  - How the task is performed.
- Scenarios
  - How does it "play out" in various situations.
    - Register for classes: All classes available
    - Register for classes: Some classes are full (wait list)
    - Register for classes: Some classes are full (I'm a senior)
- Activities (UML "Activity Diagram")
  - What activities are performed, in what order?

 Allen Holub's OO-Design Workshop www.holub.com 110

A Use-case Template (1)

1. Use-case name.
2. "Customer" contact information.
3. Executive summary.
4. Desired outcome.
5. User goals.
6. Participants/roles.
7. Dependencies to other use cases.

 Allen Holub's OO-Design Workshop www.holub.com 111

A Use-case Template (2)

8. Preconditions.
9. Inputs (forms, etc.).
10. Scenarios.
11. Activities (UML Activity Diagram).
12. Postconditions.
13. Outputs (reports, etc.).
14. Business rules (domain related).
15. Implementation notes & requirements.

The Use-Case Name

- All use cases should be named.
- Constantine recommends using a gerund followed by a direct object ("withdrawing funds" or "examining the passbook").
- convention encourages the use-case name to succinctly identify the operation being performed and the object (or subsystem) that's affected by the operation.
- Names should be user-centric, not system-centric.
  - "making a deposit" (user-centric) versus "accepting a deposit" (system-centric)

Customer Contact Information

- One of the more unpleasant experiences I've had was working with a Marketing guy who put together use cases based on a fantasy of what the customer wanted, rather than asking the customer. It unfortunately didn't occur to me that his (quite well done) use cases were created from whole cloth. The result was a lot of time and money wasted specifying a product that was of no interest to the customer whatever.

Use-case Description

- Describe what the use case is accomplishing.
- What will the user be doing while "withdrawing funds" or "examining a passbook," for example. Go into detail, but don't describe how the user might use a computer program.
- Don't mention "the system."

Example Description

- A *bank customer might make a withdrawal by filling out a withdrawal slip and presenting the slip to the teller. The teller then takes the withdrawal slip to a bank officer for approval. The bank officer checks the account balance and issues an approval, etc.*
  - Note that nowhere in this discussion have I talked about computer programs, menus, dialog boxes, etc. These sorts of implementation details are irrelevant at this level. (Though, of course, you'll need well-defined implementation details before you can code, we're not there yet).

Desired Outcome

- The work of value is performed.
- Describe the outcome here.
  - could be a report (in which case you should include an example of what the report will look like)
  - an event or condition (an employee will now receive health benefits)
  - etc.

User Goals

- What are the real goals of the user with respect to the use case?
- Goals are not the same thing as the use-case description. If the "Desired Outcome" section describes *what* the user hopes to accomplish, the "Goals" section describes *why* the user is doing it.

Roles and Actors

- An *actor* is a physical human being.
- A *role* is taken on by the actor.
- A given actor can take on multiple roles (in the same use case or in different ones).
  - This is Constantine's definition. Jacobson uses "actor" to mean "role."

Actors Are External

- Actors are outside the program.
  - Actors are hardly ever represented by objects.
    - An object makes a User Interface that is filled in by the actor. The actor is external.
- The fact that the same physical person might take on several roles at some juncture is irrelevant.
  - E.g. the *employee* and *manager* roles within a program might be filled by the same person (actor).

Specify Roles in CRC Form

- For each role, we need to establish two critical pieces of information:
  - The *responsibilities* of actor when in this role. For example, bank tellers get deposit and withdrawal requests from customers, and get approvals for withdrawals from bank officers.
  - The actor's *collaborators*—the roles with which communication is necessary. For example, a bank teller collaborates with both the customer and the bank officer, but the officer never collaborates directly with the customer.
- Add a Class name, and you have a CRC card, discussed in more depth below.

Use-Case Dependencies

- Might not exist, but that's unusual
- Various possibilities:
  - subset/combines.
  - uses/is-used-by (includes).
  - precedes/follows.
  - requires.
  - extends/is-specialization-of.
  - resembles.
  - equivalent.
- Diagram dependency relationships using a UML static-model diagram rather than "Use-Case Diagrams"

Preconditions

- What assumptions are you making about the state of the world when the use case begins?
  - For example, customers must have an account with the bank before they can withdraw money.
  - As a consequence, the "customer opening an account" use case must have been performed before the "customer withdrawing money" use case can be performed.

Inputs

- Documents used to do the work. A course catalog, for example, is necessary to register for classes. Specify where the documents come from. (What are their origins?) Was the document an output from another use case?
- Knowledge required by the actors to perform their role.
- Skill required by the actors to perform their role.
- Note that the information gleaned while executing a use case is not an input to the use case itself, it's an output.

Scenarios

- Scenarios are small narrative descriptions of someone working through the use case.
- They describe how a use case might "play out" in the real world.
- Use a fly-on-the-wall approach: describe what happens as if you're a fly on the wall observing the events transpire.
- Keep the scenarios in the problem domain (talking about how a bank, not a computer program that simulates a bank, is used):

Many Scenarios in a Use Case

- Scenarios specify several paths to success.
- E.g. Inside the “sign up for a class” use case, are:
  - I sign up for a class and get in (the “happy path”).
  - I want an elective, but I'm wait listed.
    - I'm automatically transferred and notified.
  - I'm wait listed for a required class:
    - I'm a senior, and the class is required for graduation. Freshmen in the class are dropped and notified.
- Failure conditions typically aren't scenarios.

Example Scenario

"Philip needs to make a withdrawal to buy groceries. He digs out his passbook from under the 3-foot pile of dirty socks in the top drawer of his dresser, and finds that his balance is big enough to cover what he needs, and he heads off to the bank..."

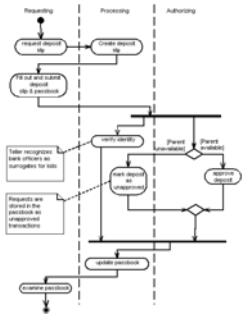
 Allen Holub's OO-Design Workshop www.holub.com 127

Activities (Workflow)

- Describe the activities that have to be performed.
  - Often the use-case description is sufficient to describe the flow of work through a simple use case ("do A, then do B, then do C").
- Use a UML "Activity Diagram" (next slide) for complex activities.
- A single Activity Diagram amalgamates *all* scenarios of a use cases.

 Allen Holub's OO-Design Workshop www.holub.com 128

An Activity Diagram



- Specifies flow, parallelism ("in any order"), and decision.

 Allen Holub's OO-Design Workshop www.holub.com 129

Postconditions

- The state of the world after the use case executes. (e.g. the account balance is now lower)

Outputs

- Documents are created during the course of the use case.
  - The output of our class-registration system is a schedule of classes, for example.
- Also list where the documents go.
  - Who will receive the document? How do you get it to them? (This last might be a use case in its own right.)
- Knowledge gained while executing the use case. What do the actors know now that they didn't know before they performed the use case?

Business Rules

- Domain-level constraints.
  - Do not describe the computer program.
- Policies that the business establishes that might effect the outcome of the use case.
- For example, "You may not withdraw more than $20 dollars within a 7-day period."

Implementation Requirements

- Customer-specified behavioral constraints on the implementation of a system. ("You must support 10,000 transactions per minute.")
- Note that some things that are called "requirements" actually aren't. (e.g. UI design).
  - A good test for whether or not a requirement is valid is to reject any so-called requirement that specifies up front something that is a natural product of the design process (UI look and feel, program organization, etc.). Such "requirements" are just bad knee-jerk design.

A Digression: Feature Lists

- It's impossible for an OO designer to work from a list of "features."
- Feature lists tend to be long, disorganized collections of poorly-thought-out ideas that some customer suggested to a salesperson off the top of their heads.
- If given one, reverse engineer to a problem statement by asking “why do you need that feature?”

Implementation Notes

- Implementation details that naturally occur to you as you work through the scenarios and workflow.
- These notes aren't bolted in concrete
  - They aren't an implementation specification; rather, they're details that will affect implementation and are relevant to the current use case.
  - They will guide, but not control the implementation-level design.

5. Design the UI

- A good UI design guides the user through the activities identified in Use-Case analysis.
  - You can't do a good UI without having done the use cases.
- Use the UI Design to verify your use cases
- Use the UI to help identify objects/classes.
- An abstract UI is fine here (post-its)

Do the UI Early (and Often)

- It tells you whether you've successfully learned the processes from your users.
- Iterate until you get it right.
- Low-Fidelity prototype == High-Fidelity conceptual model.

Use Cases Specify UI Organization

- The Activity Diagram in a use case defines the flow through the UI for that use case.
  - The UI and Use Cases are often developed simultaneously.
- See Larry Constantine and Lucy Lockwood's Book:

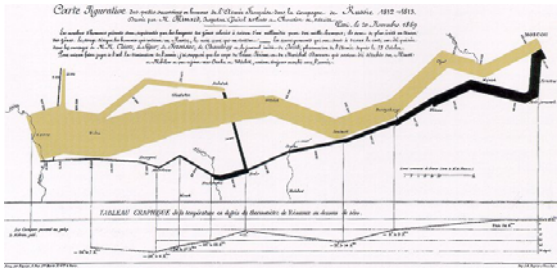
  Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design (Reading:Addison Wesley, 1999; www.foruse.com).

Good User Interfaces are Dense

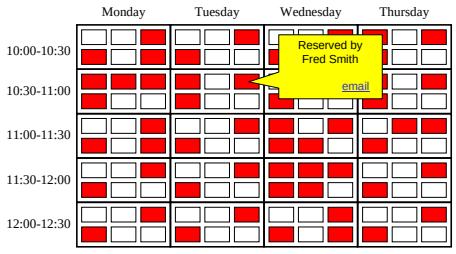
- A good UI shows you what the numbers *mean*.
- Density ≠ clutter.
  - A perfect UI shows you *everything* you need to do a task or understand a problem it in a clear and understandable fashion.
  - That's *all* it shows you.
- See Edward R. Tufte, *The Visual Display of Quantitative Information* (Graphic Press, 1983)

An Example of Density

- Napeolean's March to Moscow (from Tufte)

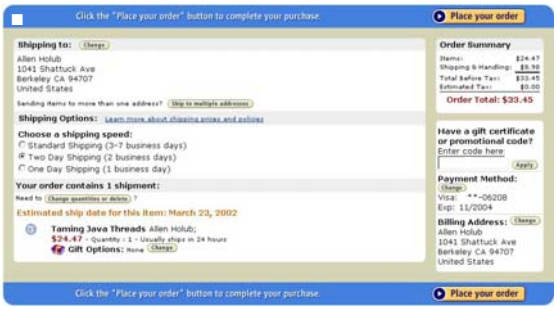


Another Dense UI: Tennis-Court Scheduling

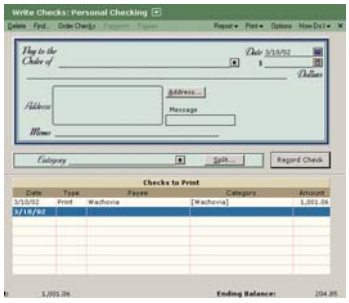


- Each `timeslot` represents a map of the six courts.
- Implies that `timeslot` objects contain `court` objects.

Yet Another Dense UI: Amazon

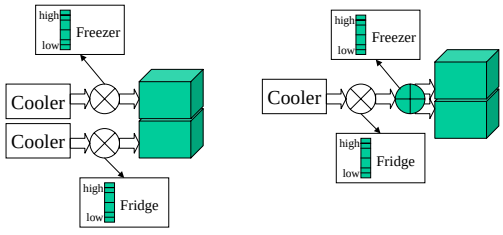


The UI Must Reflect the User's Model: Quicken



The UI Must Reflect the User's Model of the System/Process

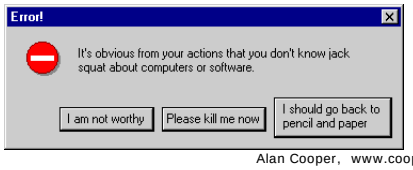
- Example from Donald Norman, *The Design of Everyday Things*.



Bad Metaphors

- File:Save
  - Assumes that the user's goal (the default operation) is to throw away the last three hour's work.
  - The metaphor is self-erasing paper.
- Moving tabs
  - The metaphor is cards magically jumping around in the box.

Assume Competence



Alan Cooper, www.cooper.com

- Assume the user is an expert.
- Don't ask twice.
  - Do what they ask, but always allow undo.
- No OKAY boxes
  - "Formatting hard disk. OKAY?"

Ask the User Before You Code

- Users must not be asked to make decisions best made by programmers.
- Find out what's best during analysis.
- If you get it wrong, refactor.

Don't Let The UI Design Guide The Process

- The UI is a natural artifact of the design process, not a precursor to it.
- A computer program is not a UI with intelligent warts hanging off of it.
  - The VB model of programming is fundamentally incorrect.

6. Develop The Dynamic Model

- A set of diagrams (one per use-case scenario) that show:
  - The objects that exist while some scenario is acted out.
  - The messages that these objects send to one another over time.

Cunningham/Beck CRC Cards

- Class / Responsibility / Collaborators
- Give up the need for global control
- OO Systems are cooperating networks of peers:
  - A program is a conversation between objects of some class.
  - Objects talk only to their collaborators, requesting operations within their area of responsibility

The Dynamic Model is a Conversation (1)

- Create CRC Cards.
- Decide on a Use case.
- Assign roles to people and hand them a CRC card.
  - The card is the "class definition." People are objects.
  - Several people might have identical cards if several objects of the same class participate in the use case.

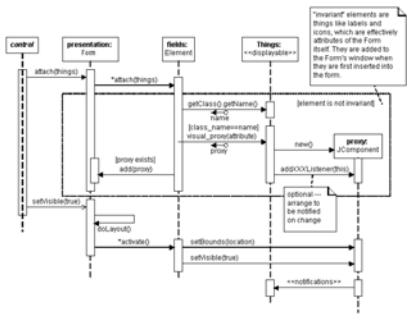
The Dynamic Model is a Conversation (1)

- Solve the problem by talking to other people:
  - You can do only those things that are listed as responsibilities on your CRC card.
  - You can only talk to collaborators.
  - You cannot give anybody any of the information that you use to do your work.
- The cards will change as the exercise progresses.
- The dynamic model is the conversation.

Dynamic Model

- Shows how the program behaves at runtime as the various scenarios play out.
  - Shows the interactions (messages sent) between *objects*, not classes.
  - Typically made up of several diagrams, each showing the objects involved in some use case and the messages sent between these objects while executing the task.
  - There should be an English description accompanying each diagram.

A Dynamic-Model Diagram



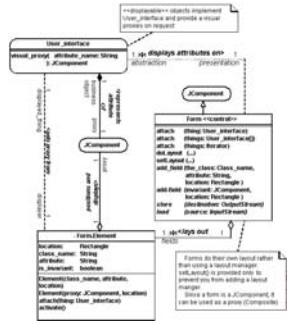
7. While Dynamic Modeling, Capture Structure in the Static Model.

- A tape recording of the conversation is the the *dynamic model.*
  - It shows how objects interact at runtime.
- The CRC cards are the *static model.*
  - It shows the relationships between classes.
- **The static model is driven by the dynamic model—it is not created first.**
  - The static-model diagram captures relationship info found in the dynamic model (e.g. who talks to whom).
  - The CRC cards change to make the conversation possible.

The Static Model

- A static-model (structure) diagram shows:
  - The classes of objects that comprise the system as a whole.
  - The relationships between objects of those classes.
    - who talks to whom and what do they say?
- You cannot create a static model in isolation.
  - It should show only those interactions that appear in the dynamic model.

A Static-Model Diagram



8. Develop The Implementation Model

- Adds implementation-specific information to all of the foregoing.
- The problem statement is expanded to describe the implementation.
- The use cases are expanded to cover commonplace failure scenarios.
- The static and dynamic models are expanded to show objects and classes that are implementation specific.

Design the Database

- Database design is one of the last things you do.
  - Schema must allow you to populate objects efficiently.
    - One table per class mandates too many joins.
    - Database often deliberately denormalized (flattened).
  - Ad-hoc queries are difficult in a database optimized for OO.
- Object/Relational mapping is a difficult problem, not yet solved effectively.
  - Automated tools do a miserable job.

When Are You Done?

- Think “architectural design” (as in buildings).
  - You show where the walls go, but not how to build a wall.
  - Plans for a skyscraper have more details than those for a house.
- You’re done when a competent programmer can implement.
  - E.g. You would never design the interface to a stock Java package like Swing.

 Allen Holub's OO-Design Workshop www.holub.com 160

9. Implement

- Implement one use case, modifying model as necessary.
  - Implementation can start as soon as you have a well-designed use case
  - Implementation and design are often parallel activities (next slide)
- Incrementally add use cases until your done, modifying model to reflect new insights, refactoring to accommodate modifications.

 Allen Holub's OO-Design Workshop www.holub.com 161

Staggered Development

- A staggered development model helps when business rules change (or are discovered) on the fly.
- Gets working code into the hands of the users quickly
- Reduces designer fatigue.

Specify Use-Case 1	Specify Use-Case 2	Specify Use-Case 3	Specify Use-Case 4	
	Model Use-Case 1	Model Use-Case 2	Model Use-Case 3	Model Use-Case 4
		Implement Use-Case 1	Implement Use-Case 2	Implement Use-Case 3	Implement Use-Case 4

 Allen Holub's OO-Design Workshop www.holub.com 162

Design and Code Reviews

Design Review Checklist (1)

- Does the design reflect the problem statement exactly? (No additions or omissions.)
- Do the use cases solve every problem specified in the problem statement without embellishment.
- Does the design realize the user's goals, even if those goals were not identified adequately in the original problem definition?
- Can an average user perform all the use cases using the UI alone and no manual?
  - Does the runtime flow through the UI match the use-case activity diagrams?

Design Review Checklist (2)

- Can an expert user read your problem statement and scenarios without you ever having to say *let me explain that*?
  - Can an expert user understand the entire design (problem statement, UI, use cases and activity diagrams, sequence diagrams, class diagrams) with you doing nothing but explaining how the notation works?
- Are the problem statement and use cases specified entirely in the vocabulary of the problem domain?
  - no computer jargon

Design Review Checklist (3)

- Is every class and object name in the model a legitimate domain term (up to the point where you start the implementation model)?
- Do associations between classes exist only when, in some dynamic-model diagram, an object of one class sends a message to an object of the other (excluding derivation associations)?
- Does the set of operations in the static model exactly mirror the set of messages used in the dynamic model?

Design Review Checklist (4)

- Does the name of every object in the dynamic model appear as a role in the static model?
- Are the message names sentences that describe what you're asking for?
  - Messages names must contain verbs!
- Has the design gotten simpler since it was first conceived?

Code Review Checklist (1)

- The phrase "let me explain how that works" is itself a defect.
- Every class and object name in the code must exactly match the design.
  - Object names are roles.
  - Message names should not be changed.
- All public methods must appear in a dynamic-model diagram.
- Method calls in the code must exactly match the message sequence shown in the dynamic model?

Code Review Checklist (2)

- All messages sent only to objects of the same class as the sender must be private.
- All fields are private. Period. No exceptions. Ever. I mean it! Really.
- When two diagrams show an object reacting differently to an identical message, is either an argument to the method or a previous message (that can force a state change) present?

Code Review Checklist (3)

- All identifiers must be words or short phrases in English.
  - The code should pass without errors through a spell checker to which the Java documentation has been added as an exception list.
- The coding style must exactly match the company style guide, even if you don't like it.

Adoption Strategies

Pilot Project, Characteristics

- Showcases OO (complicated).
- Well funded and resourced.
- Staffed with top people.
  - You're training project leaders.
- Ideally, an off-site autonomous "skunk works" project.
- Does not reimplement a legacy system.
- Does not have a fixed delivery date.
- Has an internal customer.

Pilot Project, Considerations

✓ MUST be mentored by an experienced designer.
  - Training is not sufficient. You need experience.
  - Typically a consultant.
    - In-house experts usually too busy.

✖ It takes a long time to do the first design and the results are often not ideal. The first experience you have is that OO is hard to do.

Or Start with a Design

- Hire an architect to do a complete design for you.
- Train your staff on how to read (and implement from) the design documents.
- Have them implement under the guidance of the architect.

Starting with a Design, Considerations

- ✓ Your project will be better quality, since it implements a solid design.
- ✓ The first experience you have with OO is a good one.
  - ✓ You see how fast the code goes together and how few bugs there are. People are then motivated to learn how to design themselves.
- ✖ Disadvantage: It takes longer to develop in-house design expertise.

OO Concepts and Terms

Principles vs. Maxims

- A *principle* is an inviolate precept—a fundamental truth.
  - – You do not violate your basic principles.
  - – E.g. "If it exposes implementation, it's not object oriented. Period."
- A *maxim* is a "rule of thumb."
  - – Goldberg: a guideline that is helpful in making decisions.
    - often broken when the situation requires it, but well conceived maxims tend to give you better results.
  - – E.g. An implementation-inheritance hierarchy should be no more than three levels deep.

Cohesion

- The similarity (or similarity of purpose) of the components of a class.
  - All elements of a class should be focused on achieving a common purpose. An "operating system" class, or a "graphical subsystem" class would be a bad idea because of a lack of cohesion.
- Maximize cohesion.

Coupling

- The interdependence of various components of the system on one another.
  - Global variables are a bad idea because the subroutines that use them are "coupled" to both the variable and to each other rather strongly. If you change a subroutine that modifies the variable, all subroutines that use the variable might have to change too.
- Minimize Coupling
  - A fully reusable class cannot be coupled to other classes at all (or at least, can be coupled to only a small number of other classes).

OO vs. Procedural Thinking

- Procedural: Data flows through the system, and intelligent agents act on it as it goes by. The data model is central.
- OO: A network of cooperating agents ask each other to do work. Data is hidden and data flow is minimized. The active behavior of the system is central.

It's Not the Language, Stupid

"Many people tell the story of the CEO of a software company who claimed that his product would be object oriented because it was written in C++. Some tell the story without knowing that it is a joke"

-*Adele Goldberg*

OO Structure Is Not Sufficient

- The physical structure of OO systems is just an implementation detail.
- Simply using OO structural elements (inheritance, etc.) do not make a system object oriented.
  - "I can program FORTRAN in any language"

So, What Makes it OO?

- The system is a network of cooperating intelligent agents, communicating via messages.
- Message-implementation details are unknown to the users of an object.
- The objects that have the data, do the work on that data.
  - Data is not exported or imported to or from objects.
  - Data flow is minimized.
- The system is a model of the user's notions of the problem to be solved and the domain-level solution.

What is an OO System?

- A network of intelligent agents, communicating via *messages*.
  - There's no spider in the middle of the web pulling the strands.
- An abstract model of the User's view of the problem.
  - E.g.: Model a bank.
    - Customer asks Teller for a Withdrawal Slip and fills it in.
    - Teller takes Withdrawal Slip to a Bank Officer.
    - Bank Officer authorizes transaction.
    - Teller dispenses money.

Messages

- Request the services of an object by sending a message to that object.
- The implementation of message passing varies considerably with the language.
- In Java:

```
some_object receiver = new Receiver();
receiver.message( args );
```


There's no such thing as "data"

- Fallacy: "Suppose you need to represent the data as a spreadsheet over here, and a graph over there..."
- Numbers mean something.
- There's only one optimal UI for making the meaning clear.
  - If there's more than one optimal UI, then any will do.
- The main attributes of an OO system are *operations*, not "data."

An Example: Cellular Automata

- The "Game of Life:"
  - Each cell has 8 neighbors.
  - If 1 or fewer are alive, die from loneliness.
  - If 2 exactly, cell remains unchanged.
  - If 3 exactly, cell comes alive.
  - If 4 or more are alive, die from overcrowding.
  - Otherwise, the cell is healthy.

- Cells have behavior, but are black boxes
  - implementation of behavior is unknown.
- Each cell is aware of surrounding cells only, not of program as a whole.
- Each cell can ask neighbor "are you alive?"
- Used in aerodynamics, traffic modeling, etc.

 Allen Holub's OO-Design Workshop www.holub.com 187

So What is an Object?

- An *object* is a bundle of capabilities.
  - Objects are not data structures + methods.
  - Objects communicate by messages, and *never* expose their implementation.
    - thereby eliminating the rippling effect of a change.
- Objects are defined by what they do, not what they contain.
  - See supplementary notes:
    - A String should not export the characters.
    - An Employee should not tell you its name.
    - An ATM machine should not know your account balance.

 Allen Holub's OO-Design Workshop www.holub.com 188

Strings Should Not Export Characters

- Code like this:

```
Byte[] b = my_string.getBytes();
```

  cannot be internationalized.
  - Characters must be represented as bytes.
  - Fixing this is very difficult.
    - You must find every call and modify all the code that surrounds the call.
- There's nothing you can return that's character-set independent.

 Allen Holub's OO-Design Workshop www.holub.com 189

Employees Should Not Reveal Their Identity

- What does **`getIdentity()`** return?
  - A String?
  - A Name Object?
  - An employee ID number?
  - A picture, thumbprint, retinal scan?
  - You shouldn't care.
- Use:
  - `compare_ID(anId)  // compare one ID to another`
  - `export_ID_as_XML() // returns arbitrary XML`
  - `print_name(here)  // or identity().print()`
  - `change_name()  // creates a user interface`
  - etc.

OO User Interfaces Do Achieve Traditional Goals

- OO systems separate UI code from "business logic" using an *abstraction layer*—a set of objects that build a platform-independent UI for you.
- OO systems reduce clutter by hiding even the abstract code inside low-level objects that have a "business" purpose. (e.g. **`Text`**, **`Money`**, etc.).
- See "Building User Interfaces for Object-Oriented Systems" on *www.holub.com/publications/articles*.

A Procedural ATM

1. User inserts the card, and punches in his or her PIN.
2. The ATM formulates a query of the form "give me the PIN associated with this card," sends the query to the database, and verifies that the returned value matches the one provided by the user. The ATM sends the PIN to the server as a string -- part of the SQL query -- but the returned number is stored in a 16-bit int to make the comparison easier.
3. The user requests a withdrawal.
4. The ATM formulates another query, this time: "give me the account balance." It stores the returned balance in a 32-bit float.
5. If the balance is large enough, the machine dispenses the cash, and then posts an "update the balance for this user" to the server.

Difficulties

- You can't change the database.
  - explicit: SQL is in the client.
  - implicit: code assumes an "account balance" exists.
- You can't change the algorithm.
  - fallback to credit card not supported.
- You can't change currency.
  - Consider the euro.
  - A 32-bit float won't hold some currencies.
  - You have to rewrite the code to sell your machine in another country.

An OO Solution Models a Bank

- A customer walks into a bank, gets a withdrawal slip from the teller, and fills it out. The customer then returns to the teller, identifies himself, and hands the teller the withdrawal slip. (The teller verifies that the customer is who he says he his by consulting the bank records.) The teller then obtains an authorization from a bank officer and dispenses the money to the customer.
- Not "what is your balance," but rather "am I authorized to dispense this money."

Some Rules for Objects

- **Do not ask an object for the data you need to do something, ask the object that has the data to do the work.**
- Imagine a program to be a group of intelligent, polite, and paranoid animals talking to each other along well-defined communication paths.
- Ease of construction, testing, and maintenance is inversely proportional to the amount of data that flows through the system.
- Ease of modification and debugging is inversely proportional to the number of objects you talk to and the complexity of the conversation.

Get/Set functions are **EVIL**

- The coupling relationships are too strong:
  - If you return a field, you can't change that field's type without also changing all the places where the returned values are stored.
  - If you set a value from outside, you can't change the way the value is stored without changing all the places where the value is set.
- *Might* be okay to return an object.
  - **`getName()`** is okay when
    - it returns a **`Name`** object that doesn't expose its implementation and
    - the notion of a name appears in the problem statement. (It's a "key" abstraction.)

Objects Must Create Their Own User Interfaces

- Objects that shun get/set methods must participate actively in the user interface.
- All OO user interfaces are composites of smaller UIs provided by the individual objects represented on the screen.
  - Ask: What object makes this part of the UI?
- All UI-building tools make procedural User Interfaces.
  - They will damage the structure of your program if you use them.
  - In Java, it's so easy to build a UI, you don't need a tool

Abstraction

- The technique of hiding implementation details within an object.
  - An ANSI-C `FILE` is a good example.

Encapsulation

- The containment of one object inside another in such a way that the presence of the contained object is unknown to the outside world.
- It's essential that the state data of the object be fully encapsulated and not available in *any* way.
- All variable fields must be private.

Delegation

- Often, an object that does not know how to do something will "delegate" that operation to another object.
  - That object could be a collaborator.
  - That object could be encapsulated.

Attributes

- A distinguishing characteristic of an object or class. An attribute serves to distinguish one class of objects from another, or one object of a given class from other objects of the same class.
- An attribute is not a "field."
  - e.g. A "salary" attribute might be inferred at run time from a title or pay-grade field.

Attributes (2)

- If the object didn't have a particular attribute, it would not actually be an object of that sort.
  - E.g., An "Employee" has a "salary" attribute; without the salary the class of objects should be called a "Person" or some such.
  - Attributes are not fields.
    - A "salary" could be stored as a double, as a fixed-point number, as an array of binary-coded-decimal bytes, as a string, as a character array, as a string holding the SQL you need to get the salary from a database, etc.
    - The attribute might not be stored at all — it might be computed dynamically at run time when needed.

Classes and Classification (1)

- *Classification* is a means for grouping similar characteristics or attributes.
- Classes are compile-time things, objects are runtime things.
  - For example, peon's and managers are employees and as such will support common capabilities. Managers have capabilities not supported by all employees, however, so managers form a subclass of class employee.
  - New behaviors can be added to the subclass without affecting the superclass.

Classes and Classification (2)

- Objects that have the same set of attributes are grouped together into a single class of objects.
- A *class* is effectively a description of a set of objects.
- Think "Class of objects"
  - "This class of objects can do X"
- A *superclass* (or base class) defines behavior shared by all subclasses. (Normalization)
- A *subclass* (or derived class) adds capabilities to (*extends*) or modifies behaviors of the base class.

Specialization Relationship

- The derived class is a *specialization* of the base class. The base class is a *generalization* of the derived class.
- The derived class adds specialized behavior (an Employee is a specialization of Person).

Is-A

- The derived class object *IS* a base-class object.
- You don't have derivation simply because you can say "is a:"
  - Lassie "is a" collie.
  - A Collie "is a breed."
  - But Lassie is not a breed, so there's no derivation relationship.
    - Actually, Lassie is an instance of class Dog, which has a "breed" attribute, which has the value "collie."

Inheritance

- The flip side of derivation.
- A *derived* class is said to inherit the capabilities of the base class. That is, all messages that can be handled by base class objects can also be handled by derived-class object.
  - The derived-class *inherits* all the public attributes (including methods) of the base class. It *is* a base-class object.
  - Derived-class objects can be passed to methods that expect base-class objects without difficulty, though the method cannot safely access any facility added at the derived-class level.

Interfaces

- The set of messages that objects use to communicate with each other.
- A contract that defines a set of methods that the "implementer" of the interface *must* support.
  - If you know the interfaces that a class (or object) supports, then you can send it messages defined in the interface, even if you don't know the object's actual type or class.

Implementation vs. Interface Inheritance

- Implementation Inheritance (characterized by *is* or *extends*). An implementation is inherited from the base class.
- Interface Inheritance (characterized by or *implements* or *supports*). The derived class agrees to implement capabilities defined at the base-class level but not implemented at the base-class level.
- *Use interface inheritance whenever possible*

Methods and Fields

A class definition is made up of:

- *methods*: Message-handling functions.
  1. The "method" a class of objects use to handle some request.
  2. All objects of the class share the same methods, so they are attributes of the entire class of objects.
- *fields*: The data needed by the methods to remember the object's state.
  1. Fields are an implementation detail that should always be hidden from the user of the class (are "private").

Class and Instance Variables

- A field can contain a value shared by the entire class of objects (a *class variable*)
- Or it can contain a value unique to a single object of the class (an *instance variable*)
- In Java & C++, class variables are called `static` fields for some mysterious reason.

Polymorphism (Abstract Methods)

- Characterized by "modifies."
  - A derived class modifies the way in which the base class processes a message.
  - derived class effectively replaces a base-class message handler with a different message handler for the same message.
    - A message, when received by a base-class object is handled in one way.
    - The same message, when received by a derived-class object, is handled differently, even if the sender thinks it's talking to a base-class object.
- A base-class method that can be redefined at the derived-class level is called *virtual* or *overrideable.*
- The derived-class version is called a *virtual override* or just plain *override*.

Abstract Methods & Classes

- Abstract classes cannot be "instantiated."
  - Their derived classes can be instantiated
  - They can provide implementations of some methods
    - c.f. Interfaces, which cannot provide implementations
- Abstract methods are defined, but not implemented at the base-class or interface level.
  - They must be implemented by a derived class.

Namespaces/Packages

- Organize code to facilitate team development.
- Separates programs into well-defined functional units
- The class
  ```
  com.holub.tools.Error_reporter
  ```
  is not the same as
  ```
  com.holub.tools.debug.Error_reporter
  ```
- Most languages provide an easy-to-use mechanism to choose a default package:
  ```
  import com.holub.tools.debbug.Error_reporter;
  Error_reporter reporter = new Error_reporter();
  ```


Data Flow

- Maintainability is inversely proportional to the amount of data that flows through the system.
  - Send messages, not data.
- Data that must flow through the system must be encapsulated.
- Look at an exposed attribute as a way of splitting a useful interface from an otherwise "heavyweight" object.
  ```
  some_employee.print_your_salary();
  ```
  vs.
  ```
  some_employee.what_is_your_salary().print();
  ```


A Few Books on OO

- David Taylor, *Object-Oriented Technology for the Manager* (Addison Wesley, 1991).
- Erich Gamma, et al *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley, 1995).
- Martin Fowler, *Refactoring: Improving the Design of Existing Code* (Addison Wesley, 1999).
- Armour and Miller, *Advanced Use-Case Modeling* (Reading: Addison Wesley, 2001).
- Fowler and Scott, *UML Distilled: A Brief Guide to the Standard Object* (Addison Wesley, 1999).
- Craig Larman, *Applying UML and Patterns*, (Prentice Hall, 1998)

Q&A

Allen Holub
www.holub.com