

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Personnel



- Instructor
 - Darko Marinov
- On-campus TAs
 - William Baker
 - Vytautas Valancius (Valas)
- Off-campus TA
 - Ganesh Agarwal

Texts



- Dick Hamlet and Joe Maybee, *The Engineering of Software: A Technical Guide for the Individual*
- Martin Fowler, *UML Distilled, Third Edition*
- Alistair Cockburn, *Writing Effective Use Cases*

What is (not) S.E.?



- Not just software programming
 - Individual vs. team
- Not just a process
 - Field that studies several different processes

Some definitions of S.E.



- Bauer: “The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.”
- IEEE 610: “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.”

Developing software



■ Product

- CS423 (Operating Systems)
- CS426 (Compiler Construction)
- ...

■ Process

- CS225 (Data Structure & Software Principles)
- CS427 (Software Engineering I)
- CS428 (Software Engineering II)

Activities in the process



- 427
 - requirements, architecture, design, management
- 428
 - metrics, configuration management, testing, debugging, reverse engineering

Axis of variability



- Size
- How humans interact with it
- Requirements stability/knowledge
- Need for reliability
- Need for security
- Portability
- Cost

Microsoft Word



- Size: large
- Interactiveness: high
- Requirements: frequent new features
- Reliability: moderate
- Security: low (at least used to be)
- Portability: high
- Cost: high

Space shuttle software



- Size: large
- Interactiveness: low
- Requirements: stable
- Reliability: very high
- Security: low
- Portability: low
- Cost: high

eBay software



- Size: moderate
- Interactiveness: high
- Requirements: frequent new features
- Reliability: moderate
- Security: high
- Portability: low
- Cost: low

Your example : vim

- Size: medium
- Interactiveness: high
- Requirements: changing
- Reliability: high
- Security: medium
- Portability: high
- Cost: low

Software process



- Pressman: “A framework for the tasks that are required to build high-quality software”
- Johnson: “The steps a particular group follows to develop software”
- IEEE 1074: “A set of activities performed towards a specific purpose”

IEEE 1074



■ Project Management

- project initiation
- project monitoring and control
- software quality management

■ Development

- requirements
- design
- implementation

IEEE 1074



- Post-development
 - installation
 - operation and support
 - maintenance
 - retirement
- Integral processes
 - verification and validation
 - software configuration management
 - documentation development

Defined processes



- Rational Unified Process (RUP)
 - Object oriented (OO)
 - Incremental
 - Commercial, popular
- eXtreme Programming (XP)
 - Used mostly by OO developers, not OO
 - Incremental
 - Counterculture, popular

More books



- Per Kroll and Philippe Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide*
- Kent Beck, *eXtreme Programming Explained*

Several papers



- Classics
- Latest ideas
- Lots of easy reading
- Books and papers WILL be on the exams

Purpose of course



- Be able to follow your project's process
- Be able to improve your process
- Be able to choose the right process for your project

Project



- Group project - 8 people
- Various topics
- Opportunity to practice
- Need to start thinking about projects now
- Projects from previous semesters

<http://brain.cs.uiuc.edu:8080/SEcourse/Projects>

Course info



- Wiki <http://brain.cs.uiuc.edu:8080/SECourse>
 - Announcements
 - TAs email: username ta427, host cs.uiuc.edu
- Grade: 5 homework assignments (15%),
project (35%), midterm (15%), final (35%)
- Homework 0
 - Add yourself to the list of people in the course
 - For 4 hours of credit: additional list

Next: RUP



■ Suggested reading

- Read Chapter 1 of Hamlet and Maybee
- Read Chapter 2 of Fowler
- Scan first 80 pages of Kroll and Kruchten (if you have it) or spend a few hours at
<http://www.ibm.com/software/awdtools/rup>
http://en.wikipedia.org/wiki/Rational_Unified_Process

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

TA change



- New on-campus TA
 - Jeffrey Overbey (Jeff)
- Remaining on-campus TA
 - Vytautas Valancius (Valas)
- Remaining off-campus TA
 - Ganesh Agarwal
- Fastest contact: email ta427

A definition of S.E.



- IEEE 610.12: “(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. (2) The study of approaches as in (1).”

Software process



- Pressman: “A framework for the tasks that are required to build high-quality software”
- Johnson: “The steps a particular group follows to develop software”

Purpose of course



- Be able to follow your project's process
- Be able to improve your process
- Be able to choose the right process for your project

Two (of many) processes



- Rational Unified Process (RUP)
 - Per Kroll and Philippe Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide*
- eXtreme Programming (XP)
 - Kent Beck, *eXtreme Programming Explained*

Waterfall process



- Followed in teaching, not often in practice

Requirements

Specif

SEI Capability Maturity Model



1. Initial - anything goes
2. Repeatable – tracking of process (SCM, QA, planning, requirements)
3. Defined - process defined and followed
4. Managed - measure and act on it
5. Optimizing - keep improving process

Rational Unified Process (RUP)



- Iterative
- Incremental
- Object oriented (OO)
- Commercial, popular
- Developed by Rational Software, now IBM

Best practices in RUP



- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

Components of RUP



- Artifacts – what
 - Things people make
- Roles – who
 - Roles people take
- Activities – how
 - Tasks people do
- Workflows – when
 - Order of steps people follow

Artifacts



- Vision statement [Homework 1]
- Iteration plan
- Use-case model (UML)
- Software architecture document (UML)
- Design model (UML)
- Component
- Integration build plan
- ...

Roles



- Project manager
- Architect
- Systems analyst
- Use-case specifier
- Designer
- Implementer
- Tester
- Configuration and change manager
- ...

Kinds of roles



- Manager
- Architect (technical lead)
- Business experts
 - Systems analyst, use case specifier
- Developers
 - Designer, developer, tester, CM manager

Activities



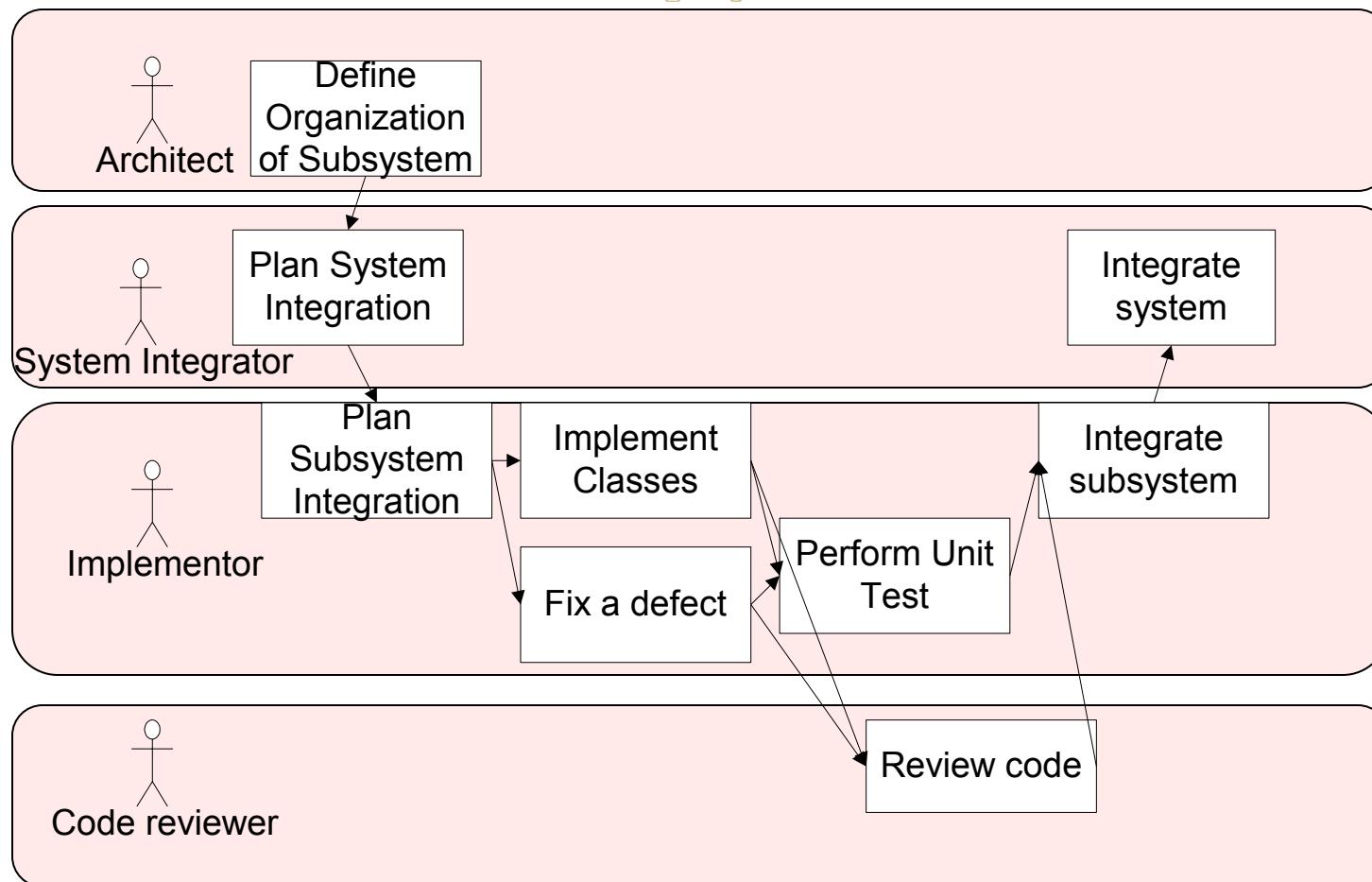
- Plan subsystem integration
- Implement classes
- Fix a defect
- Perform a unit test
- Review code
- Integrate subsystem
- ...

Workflows



- Project management
- Business modeling
- Requirements
- Analysis and design
- Implementation
- Test
- Configuration and change management
- Deployment
- Environment

Implementation workflow

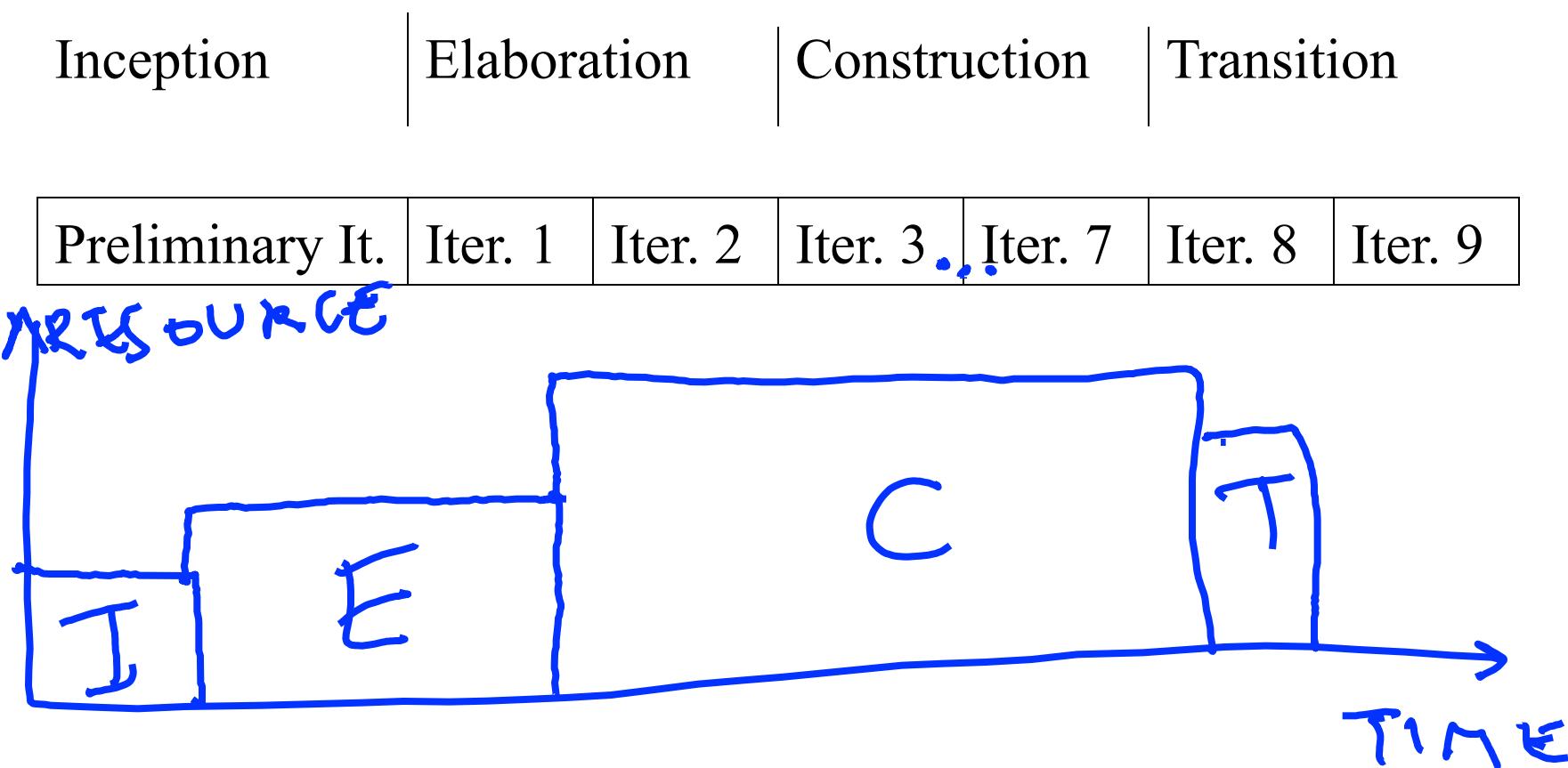


Four phases



- Inception
 - Get the idea, initial planning
- Elaboration
 - Create the architecture
 - Build skeleton system
 - Detailed planning
- Construction
 - Build the rest of the system
- Transition

Iterations



Iteration plan



- What are we going to do this iteration?
- Who is going to do it?
- What order are we going to do it?
- Do we have enough resources?

An iteration workflow



- Each iteration uses some of the standard workflows
 - Requirements, design, implementation, test
- Workflows can run in parallel
- Early iterations
 - Much project management
 - Much business modeling
 - Little implementation

Architecture



- Describes components and connections
- Component is system/subsystem
- Architect responsible for architecture
- Developers responsible for components
- Elaboration phase writes code for the architecture

Summary of RUP principles



- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

RUP is a framework



- Complicated process
- Framework - you are not expected to do everything
- Don't focus so much on the process that you forget the product!

Keys to good software



- Good people
- Understand the problem
- Understand the solution
- Enough time
- Good process

Course info



- Wiki <http://brain.cs.uiuc.edu:8080/SECourse>
 - Announcements
- Grade: 5 homework assignments (15%), project (35%), midterm (15%), final (35%)
- Homework 0, create a page if you didn't
- Homework 1
 - Project proposal, groups of 3 (may be 2 or 4)
 - Due September 21

Next: XP



■ Suggested reading

- Read what you didn't get to read
 - Chapter 1 of Hamlet and Maybee
 - Chapter 2 of Fowler
 - First 80 pages of Kroll and Kruchten (if you have it)
or <http://www.ibm.com/software/awdtools/rup>
http://en.wikipedia.org/wiki/Rational_Unified_Process
- Scan first 60 pages of Beck (if you have it) or
 - <http://www.extremeprogramming.org/>
 - <http://www.xprogramming.com/>

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Software process



- Important part of software engineering
- Johnson: “The steps a particular group follows to develop software”
- Purpose of the courses
 - Be able to follow your project’s process
 - Be able to improve your process
 - Be able to choose the right process for your project

Two (of many) processes



- Rational Unified Process (RUP)
 - Object oriented (OO)
 - Incremental
 - Commercial, popular
- eXtreme Programming (XP)
 - Used mostly by OO developers, not OO
 - Incremental
 - Counterculture, popular

Reminder



- Please read the material!
- Suggested reading for this class
 - *Extreme Programming Explained* by Kent Beck
 - <http://www.extremeprogramming.org/>
 - <http://www.xProgramming.com>

eXtreme Programming goals



- Minimize unnecessary work
- Maximize communication and feedback
- Make sure that developers do most important work
- Make system flexible, ready to meet any change in requirements

Components of a process



- How to understand a new process?
- How to compare processes?

1

Components of a process



- Artifacts – what
 - Things people make
- Roles – who
 - Roles people take
- Activities – how
 - Tasks people do
- Workflows – when
 - Order of steps people follow

Artifacts



- Metaphor (XP is evolving)
- Stories, sorted into “iterations”
- Tasks
- Unit tests
- Functional tests
- Code

Roles



- Customer [for project: someone outside of course or someone from your group]
 - Tester
- Developer
 - Coach
 - Tracker
- Boss

Activities



- Writing stories
- Planning game
- Standup meeting
- Writing tests
- Making tests work
- Refactoring
- Integrating

Overall workflow



- Customer produces sequence of “stories”
- Developers break stories into “tasks”
- Developers implement tasks one at a time, working in pairs
- Developers implement tasks by
 - Writing tests for it
 - Doing “simplest thing” to make tests work
 - Refactoring until design is simple again

XP practices



- On-site customer
- The Planning Game
- Small releases
- Testing
- Simple design
- Refactoring

XP practices

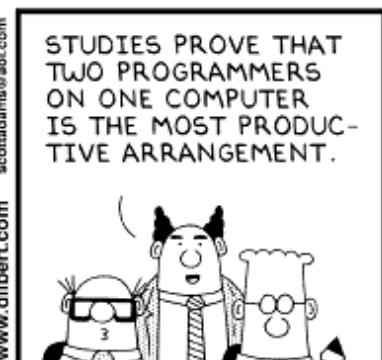


- Metaphor
- Pair programming
- Collective ownership
- Continuous integration
- Energized work
- Coding standards

Pair programming



Copyright © 2003 United Feature Syndicate, Inc.



Copyright © 2003 United Feature Syndicate, Inc.

Tests



Unit tests (xUnit)

- Test each unit of software
- Write code only if there is a unit test for it
- All unit tests must always run
- Written by developers

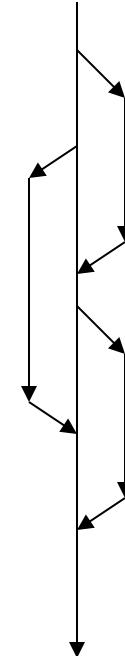
Functional tests

- Test entire package
- Written by customer

Continuous integration



- Integrate your work after each task
 - Start with official “release”
 - Once task is completed, integrate changes with current official release
- All unit tests must run after integration



Design



Design occurs when developers

- Develop metaphor
- Break story into tasks
- Decide how to implement a task
- Refactor

Simple design



- Do the simplest thing that could work
- For each design problem:
 - What are some solutions?
 - Will it work? (yes, no, maybe)
 - Pick the simplest one that might work

XP planning game



Copyright © 2003 United Feature Syndicate, Inc.

XP planning game



- Customer writes stories
- Developers estimate
 - Effort (in man-weeks)
 - Risk (high, medium, low)
- Budget for each iteration
- Customer picks enough stories to fill iterations

Planning XP project



- Write stories until they are “complete”
- Estimate and plan
- Execute and measure
- Revise plan
- Execute and measure
- Revise plan
- ...

Estimating



- If you cannot estimate a story, work on it until you can
 - Group design session
 - Prototyping
- If story is too long, break it into smaller stories
 - Divide, conquer...
 - ...and integrate

Revise plan



- Measure actual number of story-weeks implemented
- Make sure next iteration does not require more than previous iteration
- Customer can
 - Add stories
 - Revise, remove, reschedule stories
- Only developers can estimate stories

Improved estimating



- Requires a few iterations to make good estimates
- Next iterations are easier to predict than iterations that are far away
- Easy to change schedule by changing stories

Summary of XP



- Values
 - Communication
 - Simplicity
 - Feedback
 - Courage
 - Respect (XP is evolving)
- Principles
- Practices

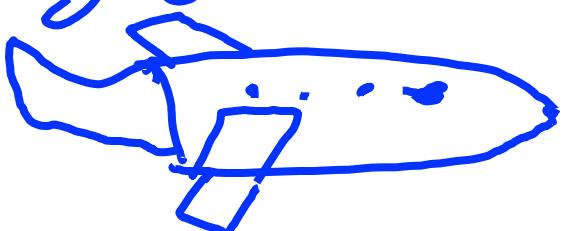
RUP vs. XP: Your opinions



- How would you compare them?
- Which one would you prefer (when)?

CHANGING : XP (vim)

BUSINESS
SETTING : RUP
: RUP?



Keys to good software



- Good people
- Understand the problem
- Understand the solution
- Enough time
- Good process

Course info



- Wiki <http://brain.cs.uiuc.edu:8080/SECourse>
 - Announcements
- Grade: 5 homework assignments (15%), project (35%), midterm (15%), final (35%)
- Homework 0, create a page if you didn't
- Homework 1
 - Project proposal, groups of 3 (may be 2 or 4)
 - Due September 21

Next: Project initiation



■ Suggested reading

- Chapters 4, 5, and 7 of Hamlet and Maybee, plus sections 2.3, 2.4, and 2.5

■ Catch-up reading

- Chapter 1 of Hamlet and Maybee
- Some overview of RUP
- Some overview of XP

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- Wiki <http://brain.cs.uiuc.edu:8080/SECourse>
 - Should be more available/reliable
 - Slides (available before lectures) and videos
 - Info for 4 hours of credit
 - Homework 0: Register on Wiki
 - 119 names (some pages with NetID), 127 enrolled
 - Homework 1: Project proposal, due Sep. 21
 - List your groups on Wiki, only a few listed

Aliases for grading



- Provide a TA with an alias/handle
 - Used to publicly and anonymously post grades
 - Makes it easy to see where you stand in class
 - 75 aliases for 107 enrolled on-campus students
- Grade: 5 homework assignments (15%), project (35%), midterm (15%), final (35%)
- Feel free to contact staff (office hours, email to schedule appointment, newsgroup)

Previous lectures



- Introduction
 - Chapter 1 of Hamlet and Maybee
- RUP overview
 - 80 pages of Kroll and Kruchten
- XP overview
 - 60 pages of Beck with Andres
- Software process: artifacts, roles, activities, workflows/principles, phases

Today: Project initiation



- To decide whether to start a project
 - Estimate cost
 - Estimate benefit
 - Do it if benefit > cost
- Cost
 - Time and effort
 - Opportunity cost
 - Future (maintenance, reduced development)

Purpose of software



- Part of a bigger system
 - Systems engineering
- Part of a business
 - Business process engineering
- Develop a product
 - Product engineering

How much to automate?



Nothing  Everything

- Air traffic control
- Inventory at store
- Drug interaction

Doctors

Why build software?



- Satisfy a need of ours
- Sell it (satisfy someone else's need)
- Make our product better
- Learn something
- Have fun

Inception phase (RUP)



- Two purposes
 - Decide whether to proceed
 - Process should help you decide
 - Convince others to proceed
 - Documents should help others decide

Inception documents (1)



■ Vision document [Homework 1]

- Why do it?
- Core requirements, key features, main constraints

■ Initial business case

- Business context
- Success criteria (revenue projections, market penetration, etc.)
- Financial forecast (budget)

Inception documents (2)



- Initial use-case model (or context diagram, or some other model)
- Initial risk assessment
- Initial project plan
 - Homework 1: e.g., prioritized list of features

Audience



- Upper management (teachers)
- Development team (other students)
- Venture capitalists/stockholders
- Users

Vision



- What problem are we solving?
- Who are the users?
- Why will the system benefit them?
- Why do we want to build this? (What is the benefit to us?)
- What will the system be like?
- What will the future be like if we are successful?

Activities in inception phase

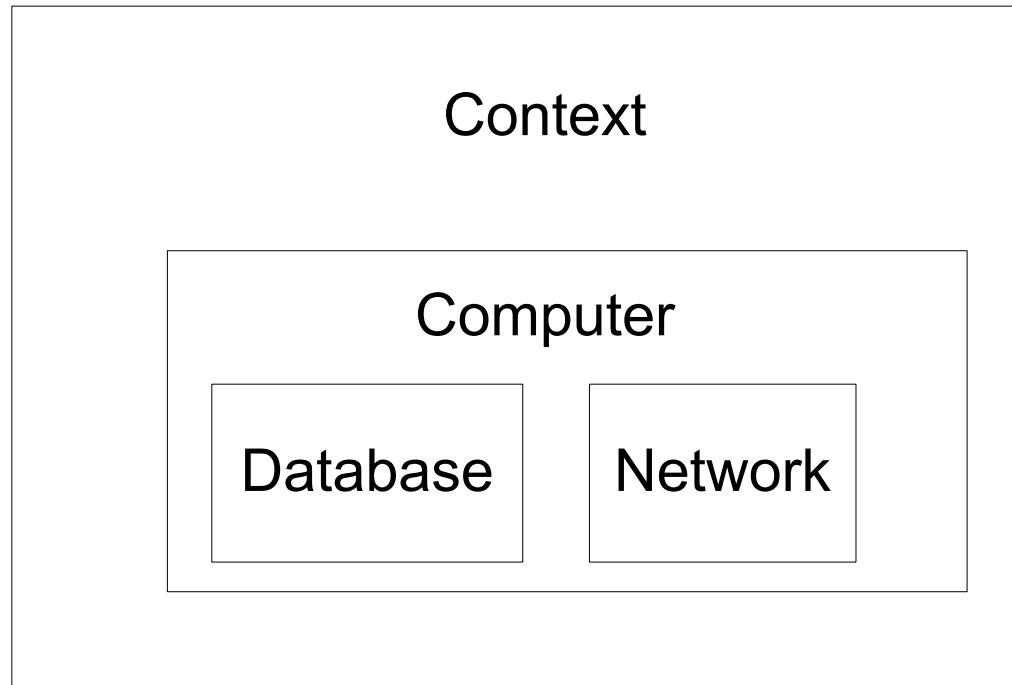


- Determine project scope and boundary conditions
- Determine critical use cases
- Prepare candidate architecture

Software in context



- What should your code not do?
- What components are in your code?



Requirements analysis



- How do you learn...
 - ...what the problem is?
 - ...what you can change?
 - ...what the computer should do?
 - ...whether you were correct?

Discovery techniques



- Reading
- Interviews
- Teams
- Creating requirements document
- Building models
- Building prototypes

Requirement document



- List requirements
- Name requirements
- Categorize requirements by
 - source
 - feature
 - subsystem
 - type

Example requirements



- Some requirements from your projects

SAVE FILE

Editor shall allow
the user to save
the text in a file.

Health-claims processing (1)



- R1) Receives health claims and supporting documents via many sources: electronically, fax, on paper.
- R2) Scanned paper and fax processed by OCR. Documents first subject to form dropout, deskewing, despeckling.
- R3) All images are logged to optical disk.

Health-claims processing (2)



- R4) Fields with low confidence levels are repaired by hand. Certain types of claims are transmitted to an offshore data entry vendor.
- R5) Match the plan and the health care provider.
- R6) Existing mainframe system processes accepted claims.

Health-claims processing (3)



- R7) Determine if claim can be paid, and how much. If there are inconsistencies, suspend the claim until a human (adjudicator) can look at it.
- R8) Adjudicator looks at documents about claim and history of client and can ask for more information, can accept claim, or can reject claim.

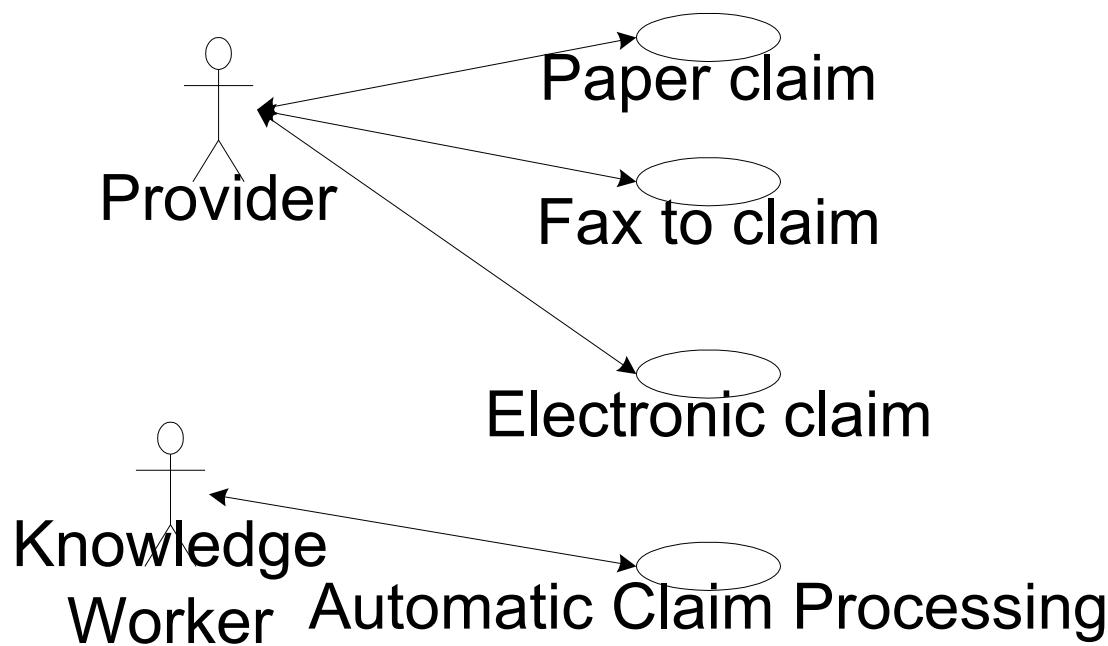
Use-case diagram



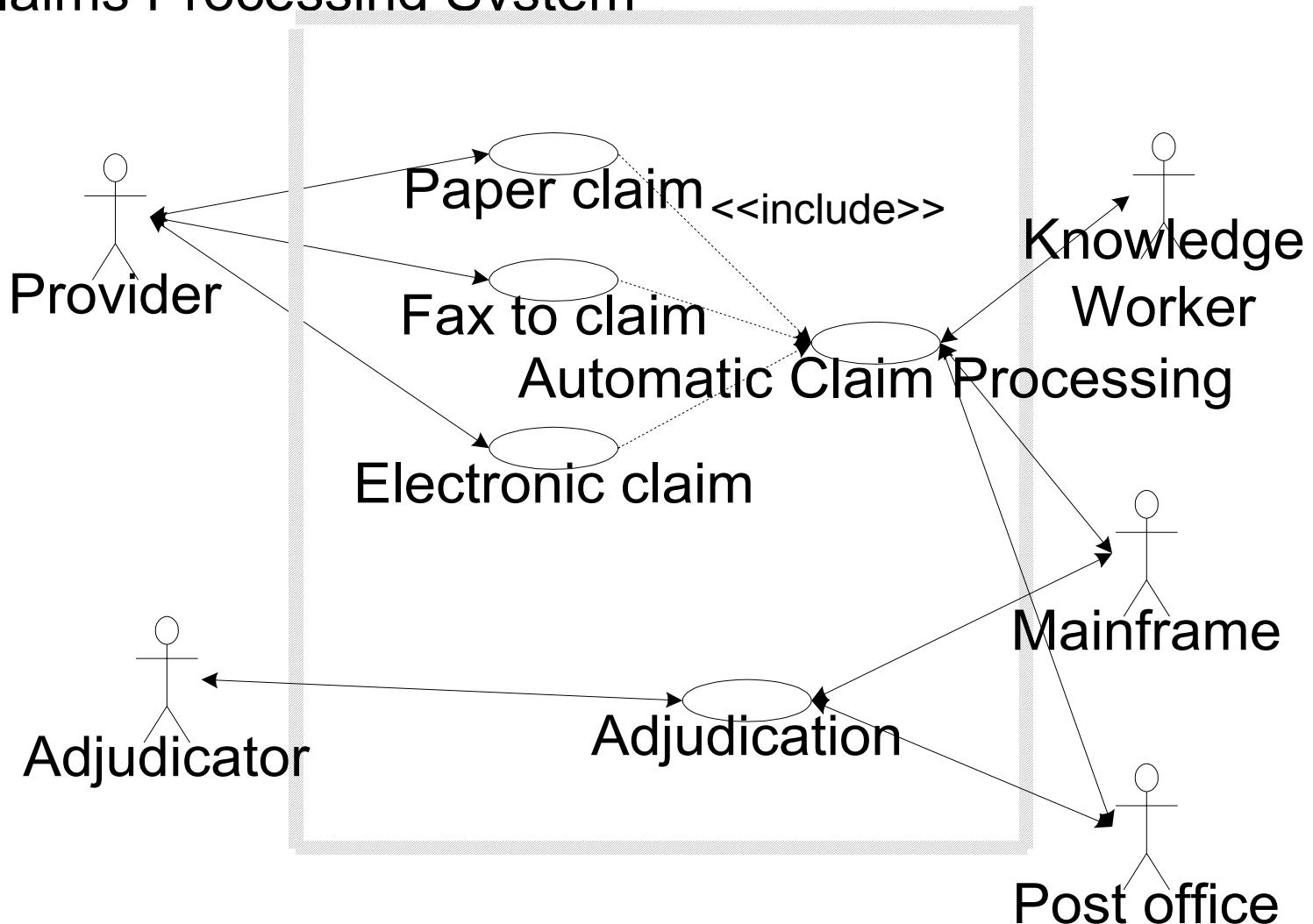
- Gives overview of system
- Shows the use cases in the system
- Components
 - Actor: Role of something or someone that interacts with System
 - Use case: Something that the System does, or that happens to the System. Always involves an actor.
 - System: The thing you are studying

Parts of claims processing

- Actors: provider, knowledge worker...
- Use cases: paper claim, claim processing...



Claims Processing System



Summary



- Goal of inception phase is to decide whether to start the project; if “yes”, then start it
- Inception phase includes a little requirements analysis and a little architecture (design)
- XP starts after the inception phase

Next: Requirements and risks



■ Reading

- Chapter 9 of Fowler's "UML Distilled, Third Edition" (on Use Cases, Ch. 3 in Second Ed.)
- Chapter 1 of Cockburn's "Writing Effective Use Cases [First Edition]"

■ Catch-up reading

- Chapters 1,2,4,5,7 of Hamlet and Maybee
- Some RUP (80p. of Kroll and Kruchten)
- Some XP (60p. of Beck)

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Previous lecture



- Project initiation
 - Decide whether to proceed
 - Cost-benefit analysis
 - Convince others to proceed, use documents
 - Vision
 - Business case
 - Initial use-case model
 - Initial risk assessment
 - Initial project plan

Requirements analysis



- How do you learn...
 - ...what the problem is?
 - ...what you can change?
 - ...what the computer should do?
 - ...whether you were correct?

- Today: (more on) requirements and risks

Requirements



- Functional requirements
 - Software inputs, outputs, and their relationship
- Non-functional requirements
 - Security
 - Reliability
 - Usability
 - Scalability
 - Maintainability
 - Efficiency
 - ...

Functional requirements (1)



- Inputs, outputs, and the relationship between them
- Use cases
- Formats, standard interfaces
- Business rules and complex formula

Functional requirements (2)



- Command language
 - The “get” command will transfer ...
- Web pages
 - Input forms, dynamic pages
- Connections to other systems
 - Files, sockets, XML, ...
- Reports, displays

Example requirements

Functional requirements in your projects

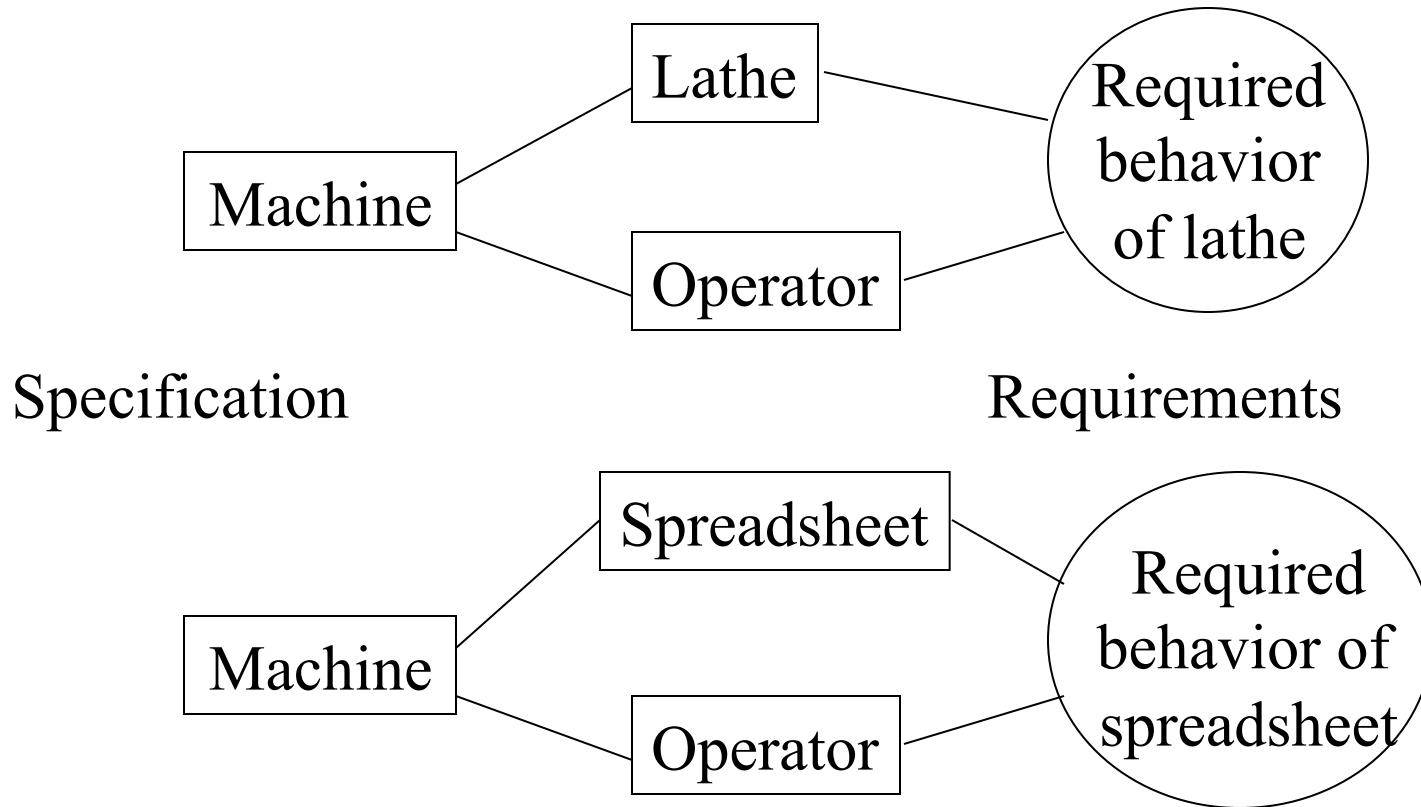
RECORD VARIANCE CONSTRAINTS

Input: - VARIANCES

Output: - ACCOUNT

UPDATE ACCOUNT
WITH THIS VARIANCE

Requirements vs. spec



Non-functional requirements



■ Performance

- Must answer a query in 3 seconds

■ Usability

- New user must be able to finish buying a book in 15 minutes
- 90% of users must say they like interface

■ Maintainability

- New programmers should be able to fix first bug in two weeks on the job

Example requirements

- Non-functional requirements in your projects

USE SQL SERVER

RESILIENT TO DISCONNECT

EASY DEPLOYMENT

USE JAVA TO IMPLEMENT

Completeness



- Functional requirements - in theory, can specify completely
- Non-functional requirements - hard to specify, can't specify completely
- Requirements should be specific, so you can tell whether you met them
- Requirements should be as precise as necessary, but no more

Comparison of requirements



■ RUP

- Use cases
- Requirement document that covers non-functional requirements

■ XP

- Stories
- On-site customer

Risks



- Technical risks
 - Project risks
 - Business risks
-
- Success does not require winning big, but avoiding failure

What can go wrong?



- Problem harder than we thought
- We waste time, take too long
- Customers don't know what they want
- System is too slow
- Stock options become worthless, developers quit
- Developers get bored and quit
- Developers like new technology; it doesn't work
- Customers run out of money and kill the project

Example risks

■ Risks in your projects

- * PEOPLE LEAVING/SLACKING
- * KILL PROJECT
- * INTEGRATION PROBLEM
 - DESIGN
 - ~~KEEP INTEGRATING~~ ~~RUP~~ ~~XP~~
- * TOO LATE
 - LIST OF FEATURES
 - START SLOW AND GROW
 - (OVER)ESTIMATE

Common risks



- Projects get killed for same old reasons
- Use database of problems to identify risk
 - Assess risks
 - Avoid risks
 - Monitor risks you can't avoid
 - Manage risks

Biggest risks



- Process should address biggest risks:
 - Wrong requirements
 - Iterative development
 - Work closely with customer
 - Constantly changing requirements
 - Manage changes
 - Keep schedule up to date
 - Inadequate schedule
 - Keep schedule up to date

Risk management



- Risk assessment: a document listing risks, their likelihood, and their severity
- Decide whether you are going to try to avoid this risk or just to monitor it

Managing risk in RUP



- Iterations alleviate risk
 - Feedback
 - Chance to try out new technology
- Architecture should address known risks
- Rank use cases by customer priority and risk
- Management is responsible for non-technical risk

Managing risk in XP



- Iteration
- Do the simplest thing that could possibly work (DTSTTCPW)
- Customer picks most important stories
- Developers estimate time
 - Stories that can't be estimated or have long estimates are risky; developers must work (prototype) to make reliable estimate

Differences (1)



- Risk: Later iterations need more sophisticated design than first ones
- RUP: Elaboration phase should consider all use cases and should select ones that are risky
- XP: Keep design clean and simple and change it if necessary

Differences (2)



- Risk: As developers leave the project, new ones must learn the system
- RUP: Document the architecture and key use cases
- XP: Use pair programming to teach new developers the system and to make sure that knowledge of the system is spread as widely as possible (diversify)

Example managing risk



■ Managing risk in your projects

Next: Use Cases



- Read chapters 2-5 of Cockburn
- Reading is IMPORTANT; probably repeated enough times by now

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Previous lectures



■ Project initiation

- Decide whether to proceed, cost-benefit
- Convince others to proceed, use documents
 - Vision, use cases, risks, project plan [Homework 1]

■ Risks

- Technical, project, business

■ Requirements

- Functional vs. non-functional

Requirements



- Functional requirements
 - Software inputs, outputs, and their relationship
- Non-functional requirements
 - Security, reliability, usability, scalability, maintainability, efficiency...
- Today: Use cases
 - High-level system description



System description



- Typical description has two parts
 - Data
 - Operations on that data
- Three kinds of descriptions
 - Requirements
 - Specification
 - Design

Many notations



■ UML

- Use cases
- Class diagram
- State diagram

■ Hamlet and Maybee

- Finite state machine, data flow diagram
- Prolog
- Pseudocode

Many purposes



- Communicate to
 - User
 - Developers
 - Boss
- Complete - lots of detail
- Easy to read - less detail

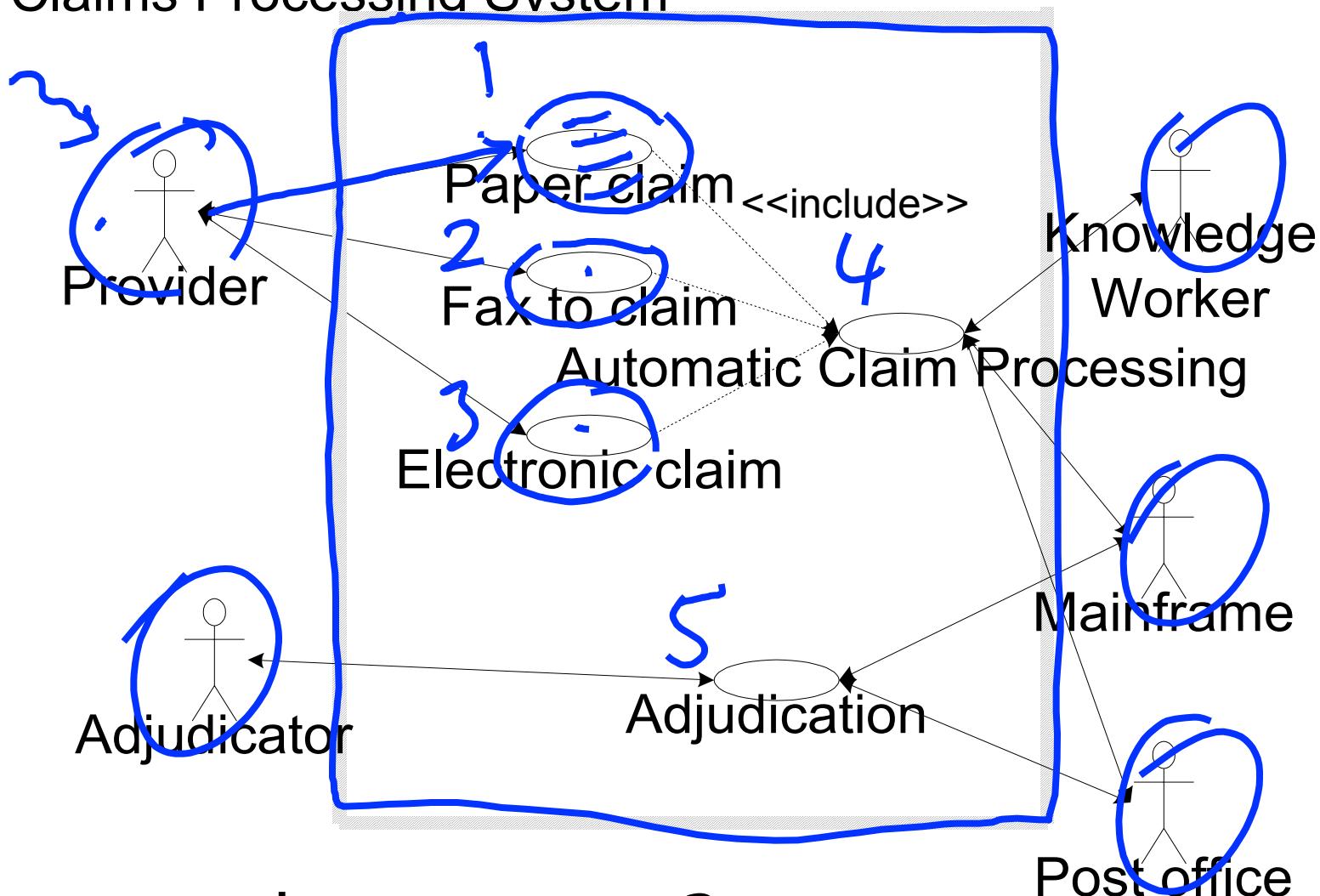
Writing Effective Use Cases



- Author: Alistair Cockburn
- Know the inside cover, front and back
- More info on Web (Wikis)
 - <http://www.usecases.org> (some broken links)
 - <http://c2.com/cgi/wiki?UseCases> (a bit old)
- Papers
- Discussion boards

USE CASE DIAGRAM

Claims Processing System



Where are the use cases?

Use-case diagram



- Shows actors and use cases
- Shows system scope and boundaries
- Not a description of use cases themselves

Use cases



- Text - a form of writing
- Describes the system's behavior as it responds to a request from an actor
- Many kinds of use cases
 - Brief / detailed
 - Requirement / specification / design

Sequence of events



- Use cases describe the sequence of events that happen when the system responds to a request
 - Can describe alternatives
 - Can describe errors

Use cases are text



- Use cases should be easy to read
 - Keep them short
 - Tell a story
 - Write full sentences with active verb phrases
 - Make the actors visible in each sentence
 - Variation: Actor explicitly first, followed by a colon

Many variations of writing



- User stories 
- Actor-goal list
- Use case briefs
- Casual use cases
- “Fully dressed” use cases

SUMMARY

USER-GOAL

Actor-goal list

WAVES
SUBFUNCTION

Actor	Task-level goal	Priority
Provider	Submit paper claim	1
Provider	Submit fax claim	2
Provider	Submit electronic claim	3
Adjud.	Adjudicate problem	2

YOUR EXAMPLE

GROUP OF ?
STUDENTS
STUDENT
STUDENT

SCHEDULE A MEETING
UPDATE AVAILABILITY
FIND FREE TIME

Use case briefs



Actor	Goal	Brief
Provider	Submit paper claim	Submit claim on paper, which is converted into electronic form, and either paid or sent for adjudication.
Provider	Submit fax claim	Submit claim by fax, which is processed by OCR and either paid or sent for adjudication.
Adjudicator	Adjudicate failed claim	Decide whether a questionable claim should be paid by the mainframe payment system or rejected.

Brief (casual) version of Submit Fax Claim



The Provider submits a claim by fax. The claim processing system will log the image to optical disk, apply form dropout, deskewing, despeckling, and then process it using OCR. Knowledge workers will repair any fields that seem to be in error. The claim will then either be paid (using existing mainframe processing system) or sent to adjudication.

Detailed (fully dressed) version of Submit Fax Claim

- Primary actor: Provider **ACTOR**
- Goal in context: Pay legitimate claims while rejecting bad ones
- Scope: Business - the overall purchasing mechanism, electronic and nonelectronic, as seen by the people in the company
- Level: Summary
- Stakeholders and interests:
- Provider: Wants to be paid for services rendered
 - Company: Wants to cut costs and avoid fraud
 - Government regulatory agency: *laws are obeyed*
- Preconditions: none
- Minimal guarantees: Pay only certified providers, pay only for services that are covered by plan, do not pay if there are obvious mistakes

Main success scenario:



Trigger: Claim submitted by fax

1. Provider: submit claim by fax
2. Claim system: drop forms, deskew, despeckle, use OCR to convert to electronic form
3. Claim system: check claim to make sure it is legal
4. Mainframe payment system: pay claim

Extensions:

- 2a. Some fields have low confidence: Knowledge worker corrects
- 3a. Claim is not valid: Send to adjudication

Your example (1)



Use case:

Scope:

Level:

Primary actor:

Goal:

Stakeholders and interests:

Preconditions:

Trigger:

Minimal guarantees:

Success guarantees:

Your example (2)



Main success scenario:

Extensions:

Use cases and requirements



- An important part of requirements
- Help requirement traceability
- Help manage requirements

Requirements



- Use cases
- Stakeholders - people who care
- Business case - cost of project, time to complete project
- Interfaces with outside systems
- Technology requirements
- Ease of use, ease of maintenance, throughput and response time

Traceability



- Traceability - the ability to trace the effect of a requirement and determine who caused it
 - Why do we have this requirement? Who wrote it?
 - How is this requirement met?
 - What requirement caused this design?

Manage requirements



- Must agree to change in requirements
 - Usually increases price
 - Must be reviewed
- Make sure each part of design is due to a requirement
- Analyze problems: what was the root cause of this bug?

Scenarios and use cases



- Scenario is concrete and detailed
 - Names of people
 - \$ values, particular dates, particular amounts
- Scenario is a test case
- Use case is a contract and collects several scenarios

Goals and use cases



- Actor has a goal for the use case
- System forms subgoals to carry out its responsibility
- Goals can fail
- Use case describes a few ways for carrying out the goal, and a few ways of failing

When use cases don't work



- Compilers
 - One use case - compile a program
- Despeckler
 - One use case - remove speckles
- No interaction
- Complexity is caused by
 - Input format
 - Transformation

Summary



- Use cases are useful, but not perfect
- Many ways to write use cases
- Big projects need big use cases
- Use the simplest way you can!

Next: Planning



- Chapter 8 of Hamlet and Maybee
- RUP: Chapters 12 and 14 of Kroll & Kruchten
- XP: Chapter 12 of Beck & Andres or
Planning Game (note XP evolution) from say
<http://c2.com/cgi/wiki?PlanningGame>

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Use cases



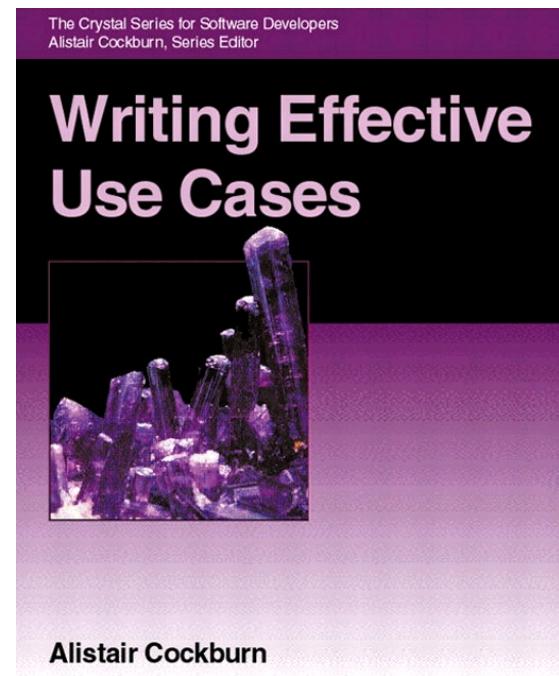
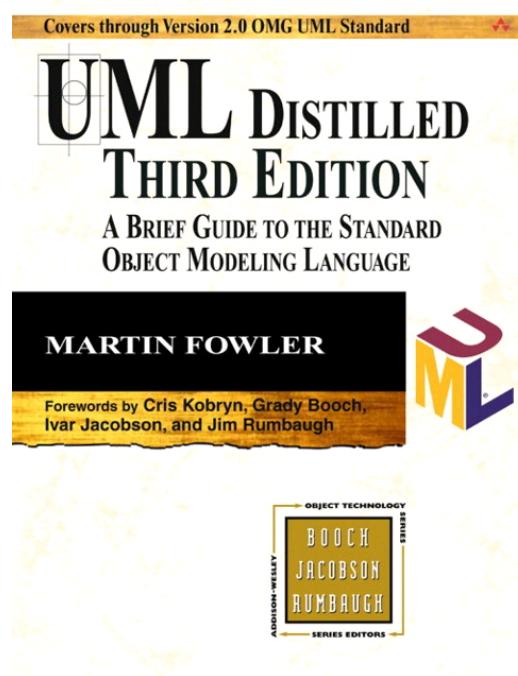
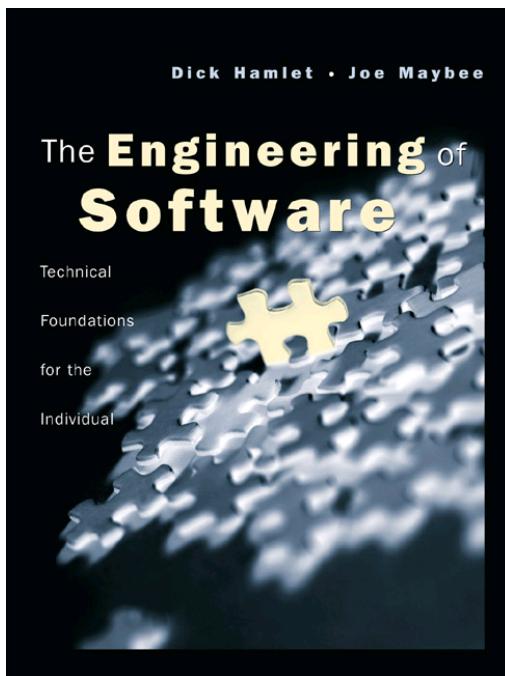
- Started discussing in the previous lecture
- Continuing today
- Why didn't we finish?
 - Different reading expectations
 - Read material before or after the lecture?
 - Unknown domain of health claim processing
 - Let's develop use cases to understand the domain

Reading mentioned from L1

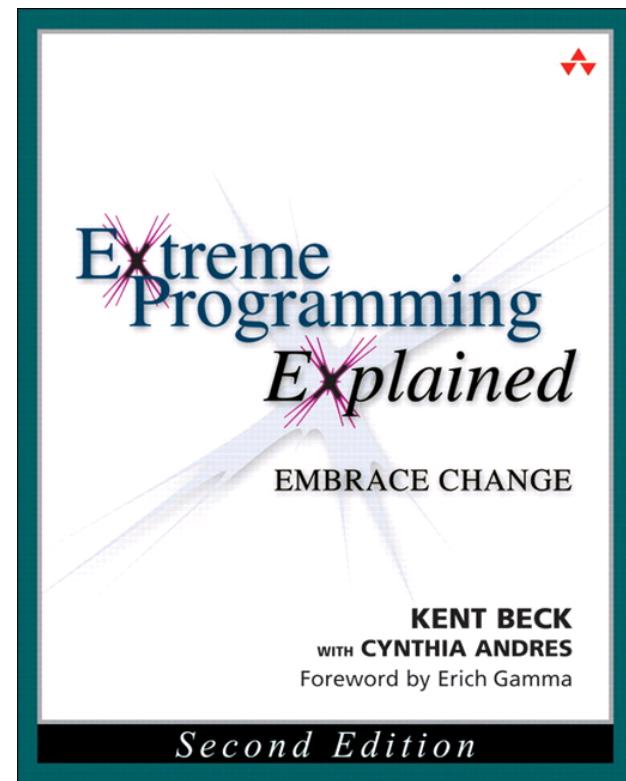
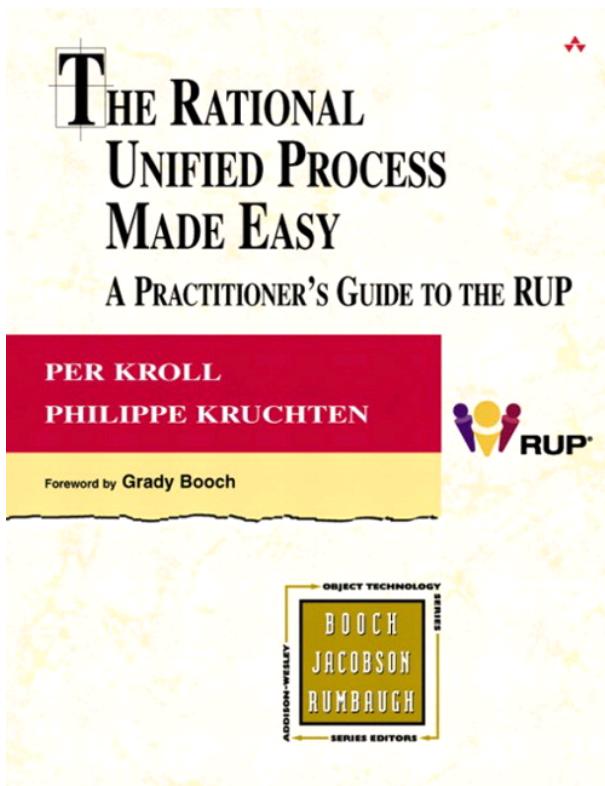


- Three main books
 - One book each for RUP and XP
 - Several additional papers
-
- LOTS of (easy) reading
 - Books and papers WILL be on the exams

Three main books



Two process-specific books



Reading for each class



- Shown in the last slide (Next Lecture)
- Listed on Wiki (for three main books)
- For use cases
 - L4: Chapter 1 of Cockburn
 - L5: Chapters 2-5 of Cockburn
- L5: Reading is **IMPORTANT**; probably repeated enough times by now

Reading before or after class?



- You are expected to do the reading
 - Books and papers WILL be on the exams
- If you read before the lectures
 - Lectures can get into advanced material
 - Discussion about topics (help with HW)
- If you read after the lectures
 - Lectures can cover only the basics

Reading assigned so far

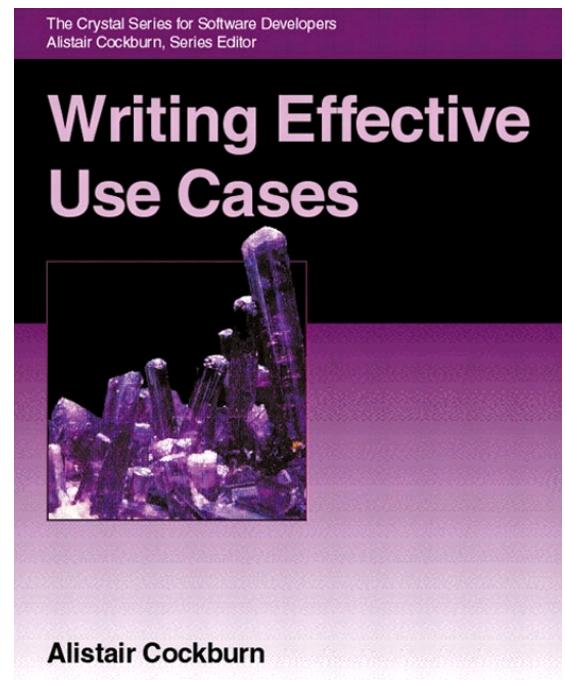


- Chapters 1,2,4,5,7 of Hamlet and Maybee (110 pages)
- Some RUP (80p. of Kroll and Kruchten)
- Some XP (60p. of Beck)
- Chapter 9 of Fowler (10 pages)
- Chapters 1-5 of Cockburn (80 pages)
- 350 pages running total

Back to use cases



- “Writing Effective Use Cases” by Alistair Cockburn
- Chapters 1-5
- Know the inside cover, front and back



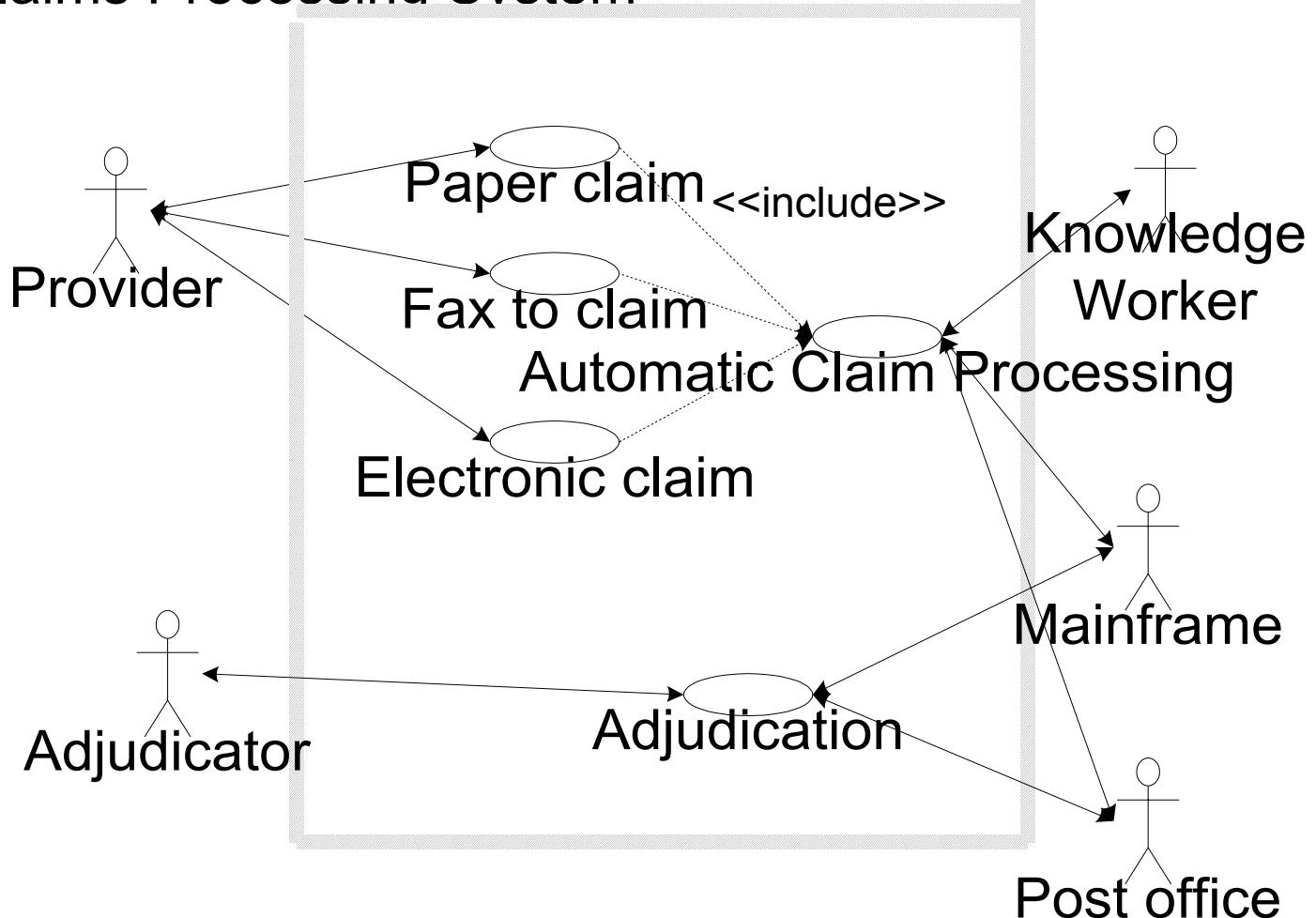
Use cases



- Text - a form of writing
- Describes the system's behavior as it responds to a request from an actor
- Many kinds of use cases
 - Brief / detailed
 - Requirement / specification / design

Unknown domain

Claims Processing System



Start from something known



- Let's describe how dentist visits go

Joe's tooth hurts, and so he wants to visit his dentist Sam. Joe makes an appointment. Joe goes to Sam. Joe reports the problem. Sam diagnoses the problem and does something to the tooth. Joe is happy ☺ and goes home.

- 
- Use case: Patient visits a dentist
 - Primary actor: Patient
 - Goal: Get tooth fixed
 - Stakeholders:
 - Dentist
 - Hospital
 - Insurance company



■ Main success scenario:

1. Patient visits the office.
2. Patient describes the problem.
3. Dentist diagnoses and repairs.

Example use case



- Use case 2: Dentist gets paid
 - Scope: Business process for dentists
 - Level: Summary (kite)
 - Actor: Dentist
 - Goal: Receive payment for the service
1. Dentist files a claim
 2. Insurance company processes the claim
 3. Insurance company sends a check
 4. Dentist tells the patient how much he needs to pay

Use cases are text



- Use cases should be easy to read
 - Keep them short
 - Tell a story
 - Write full sentences with active verb phrases
 - Make the actors visible in each sentence
 - Variation: Actor explicitly first, followed by a colon

Uses case are sequences



- Use cases describe the sequence of events that happen when the system responds to a request
 - Can describe alternatives
 - Can describe errors

Many variations of writing



- User stories
- Actor-goal list
- Use case briefs
- Casual use cases
- “Fully dressed” use cases

Actor-goal list



Actor	Task-level goal	Priority
Provider	Submit paper claim	1
Provider	Submit fax claim	2
Provider	Submit electronic claim	3
Adjud.	Adjudicate problem	2
Patient	Get tooth fixed	100
Dentist	Get paid	200

Use case briefs



Actor	Goal	Brief
Provider	Submit paper claim	Submit claim on paper, which is converted into electronic form, and either paid or sent for adjudication.
Provider	Submit fax claim	Submit claim by fax, which is processed by OCR and either paid or sent for adjudication.
Adjudicator	Adjudicate failed claim	Decide whether a questionable claim should be paid by the mainframe payment system or rejected.
Dentist	Gets paid	Dentist submits claim, H.S. 10. and patient split w/

Brief (casual) version of Submit Fax Claim



The Provider submits a claim by fax. The claim processing system will log the image to optical disk, apply form dropout, deskewing, despeckling, and then process it using OCR. Knowledge workers will repair any fields that seem to be in error. The claim will then either be paid (using existing mainframe processing system) or sent to adjudication.

Casual form for Dentist Gets Paid



Dentist files a claim. Insurance company processes the claim and sends a check. Dentist tells the patient how much he needs to pay and receives one more check.

Detailed (fully dressed) version of Submit Fax Claim



Primary actor: Provider

Goal in context: Pay legitimate claims while rejecting bad ones

Scope: Business - the overall purchasing mechanism, electronic and nonelectronic, as seen by the people in the company

Level: Summary

Stakeholders and interests:

Provider: Wants to be paid for services rendered

Company: Wants to cut costs and avoid fraud

Preconditions: none

Minimal guarantees: Pay only certified providers, pay only for services that are covered by plan, do not pay if there are obvious mistakes

Main success scenario:



Trigger: Claim submitted by fax

1. Provider: submit claim by fax
2. Claim system: drop forms, deskew, despeckle, use OCR to convert to electronic form
3. Claim system: check claim to make sure it is legal
4. Mainframe payment system: pay claim

Extensions:

- 2a. Some fields have low confidence: Knowledge worker corrects
- 3a. Claim is not valid: Send to adjudication

Extensions for our Use Case



Use case: Dentist gets paid

Precondition: Dentist has insurance info for the patient

Main success scenario:

1. Dentist files a claim
2. Insurance company processes the claim and sends a check
3. Dentist tells the patient how much he needs to pay

Extensions:

- 2a. Insurance company goes bankrupt
- 2b. Deny the claim (incorrect procedure)
- 2c. Check gets lost
- 3a. Dentist tells the patient that the service was overpaid and sends a check to the patient
- 3b. Patient does not pay

Your example (1)

Use case:

TITLE

Scope:

ORGANIZATION



SYSTEM



Level:

PERSON OR SYSTEM

COMPONENT

Primary actor:

Goal:

Stakeholders and interests:

PEOPLE OR SYSTEMS INTERESTED

Preconditions:

IN USE CASE ASSUMPTIONS

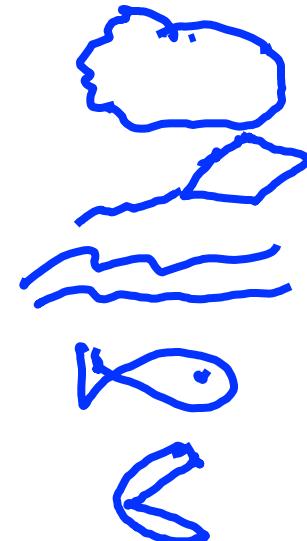
Trigger:

START

Minimal guarantees:

END

Success guarantees:



Your example (2)



Main success scenario:

STEPS

Extensions:

ALTERNATIVES — SUCCESSFUL
— FAILING

Use cases and requirements



- An important part of requirements
- Help requirement traceability
- Help manage requirements

Requirements



- Use cases
- Stakeholders - people who care
- Business case - cost of project, time to complete project
- Interfaces with outside systems
- Technology requirements
- Ease of use, ease of maintenance, throughput and response time

Traceability



- Traceability - the ability to trace the effect of a requirement and determine who caused it
 - Why do we have this requirement? Who wrote it?
 - How is this requirement met?
 - What requirement caused this design?

Manage requirements



- Must agree to change in requirements
 - Usually increases price
 - Must be reviewed
- Make sure each part of design is due to a requirement
- Analyze problems: what was the root cause of this bug?

Scenarios and use cases



- Scenario is concrete and detailed
 - Names of people
 - \$ values, particular dates, particular amounts
- Scenario is a test case
- Use case is a contract and collects several scenarios

Goals and use cases



- Actor has a goal for the use case
- System forms subgoals to carry out its responsibility
- Goals can fail
- Use case describes a few ways for carrying out the goal, and a few ways of failing

When use cases don't work



- Compilers
 - One use case - compile a program
- Despeckler
 - One use case - remove speckles
- No interaction
- Complexity is caused by
 - Input format
 - Transformation

Summary



- Use cases are useful, but not perfect
- Many ways to write use cases
- Big projects need big use cases
- Use the simplest way you can!

Next: Planning



- Chapter 8 of Hamlet and Maybee
- RUP: Chapters 12 and 14 of Kroll & Kruchten
- XP: Chapter 12 of Beck & Andres or
Planning Game (note XP evolution) from say
<http://c2.com/cgi/wiki?PlanningGame>
- READING IS IMPORTANT
- Feel free to ask course staff about CS427

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Questions on reading



- Reading before or after?
 - Voting declares the winner: before
- Overlap of reading and lectures?
 - Lectures typically cover less
- How much to read?
 - See exams from previous years
 - Use “History” on top of Wiki pages

Homework 1



- Postponed for Tuesday, September 26
 - Fair warning: don't wait until the deadline
- Deliverables
 - Vision document (lecture 4)
 - Use case model (lectures 6 and 6e)
 - Risk assessments (lecture 5)
 - Project plan (lecture 7, today)

Planning quotes



- Anonymous: “Nobody plans to fail. They just fail to plan.”
- Eisenhower: “Plans are nothing. Planning is everything.”
- Moltke: “No battle plan survives contact with the enemy.”

Who plans?



- Managers
- Developers?
- Testers??
- XP
 - 1st edition: planning is a cooperative “game” between the customer and the developers
 - 2nd edition: not called “game” any more

Management



- Decides what to build
 - Product scope and requirements
- Gets and organizes people
- Selects process
- Plans project
- Evaluates progress
- Managers vs. leaders

Manager vs. architect



Manager

- Pizza
- Evaluation
- New computers
- Top management
- Builds team
- Schedules

Architect

- Requirements
- Client-server vs. web-based
- Design reviews
- Tough technical decisions

Planning



- How much will it cost?
- How long will it take?
- How many people will it take?
- What kind of people will it take?
- What might go wrong?
- What can we do to minimize chance of failure?

Examples



- How much would we save if we used Crystal Reports to generate our reports?
- How much would we save if we outsourced the project to Smith Brothers Software?

Planning



- Scoping - understand problem and the work that must be done
- Estimation - time and effort
- Risk - what can go wrong? What can we do about it?
- Schedule - resources and milestones
- Control strategy - how can we tell when things are going wrong?

Planning is easy



- Figure out all the tasks
- Figure out how long each task will take
- Figure out who can do the work
- Assign workers to tasks to produce shortest schedule

Planning is hard



■ Tasks

- Depends on requirements, design
- Meetings, emergencies, learning, helping others

■ Estimation

- Compare with past
- Improves with practice
- Simple tasks easier to estimate than complex

Principles of planning



- Estimates should be made by people who do the work
- Estimates for small items are more accurate than for large items
- Learning to estimate requires practice
 - HW1 is an opportunity to start practicing
 - You won't be graded on accuracy of plans

Yet another book?



- “Facts and Fallacies of Software Engineering” by Robert Glass
 - 55 facts and 10 fallacies
 - No, you don’t have to read it
- One fact on importance of estimation
 - “One of the two most common causes of runaway [software] projects is poor estimation.”
 - What do you think the other cause is?

Facts on when and who



- “Estimation occurs at wrong time.”
 - Most estimates are made at the beginning of a project, before requirements are defined and thus before the problem is understood.
- “Estimation is done by the wrong people.”
 - Most software estimates are made either by upper management or by marketing, not by the people who will build the software or their managers.

Facts on flexibility



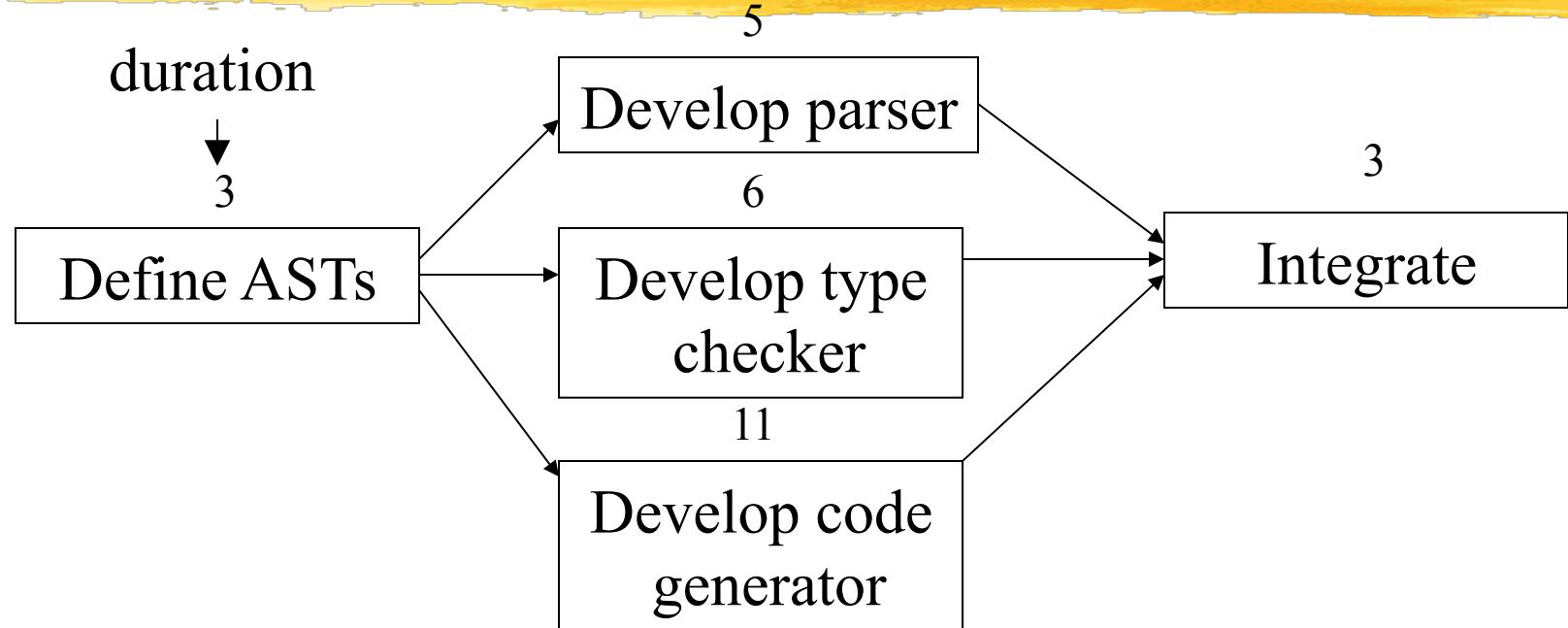
- “Software estimates are rarely corrected as the project proceeds.”
 - Thus, those estimates done at the wrong time by the wrong people are usually not corrected.
 - How do XP and RUP address this problem?
- “It is not surprising that estimates are bad. But we live and die by them anyway!”
 - There is little reason to be concerned when projects do not meet estimated targets. But people are concerned anyway.

Scheduling



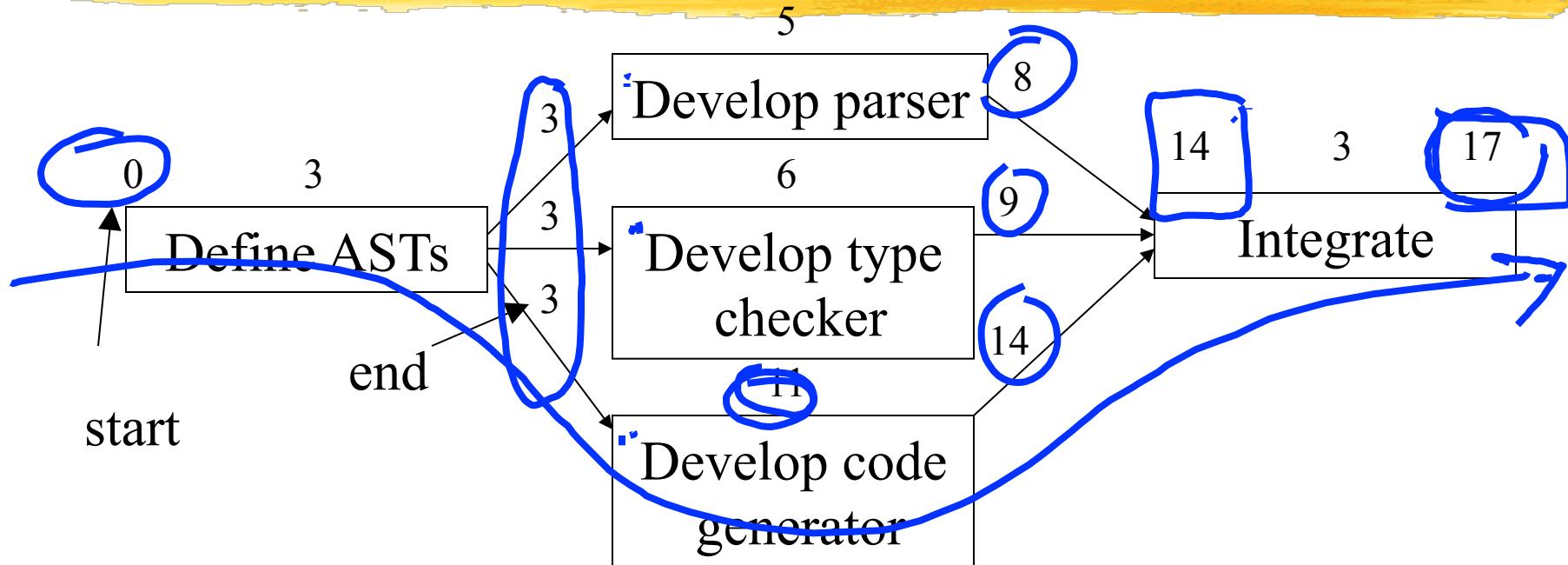
- Determine needed resources for each task
 - People and skills
 - computers, testing system
- Schedule tasks
 - Some tasks depend on others
 - Some tasks compete for resources

PERT charts (1)



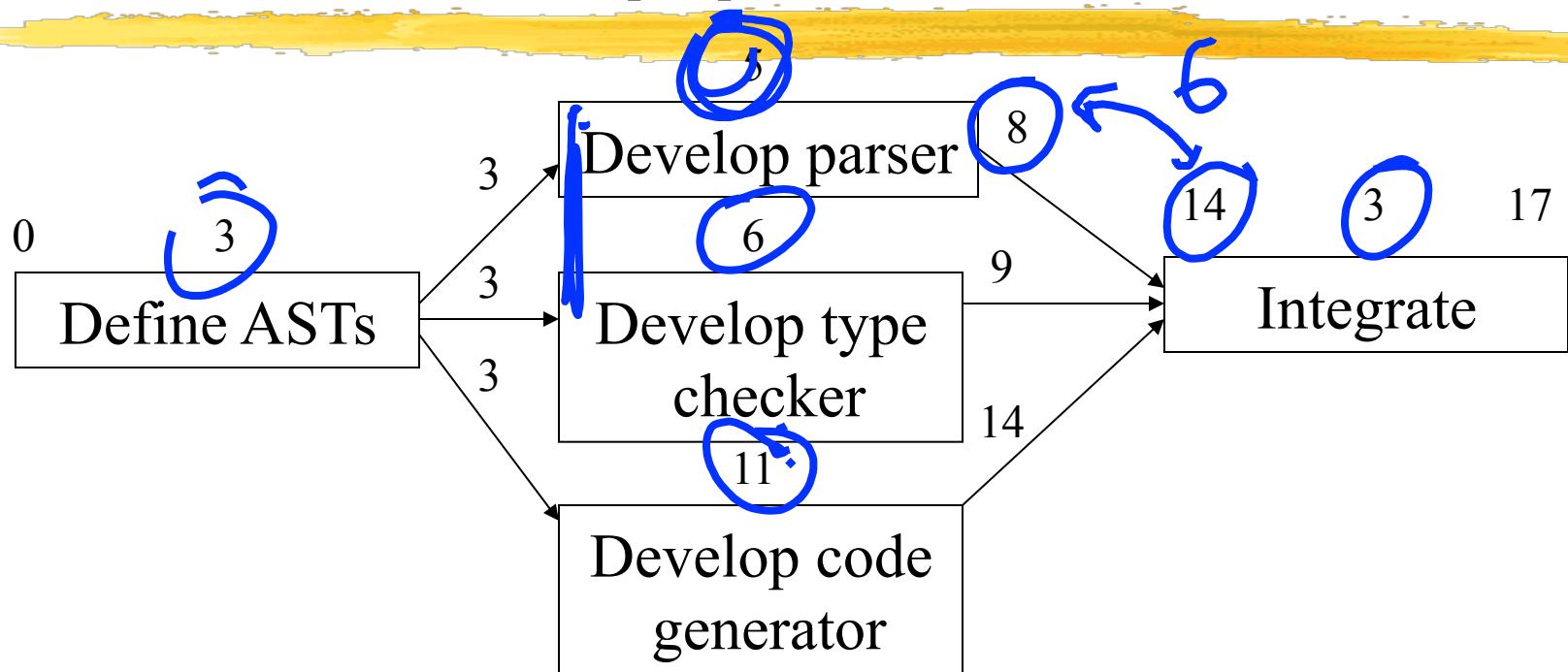
- Represent tasks and their duration
- Manage dependency between tasks

PERT charts (2)



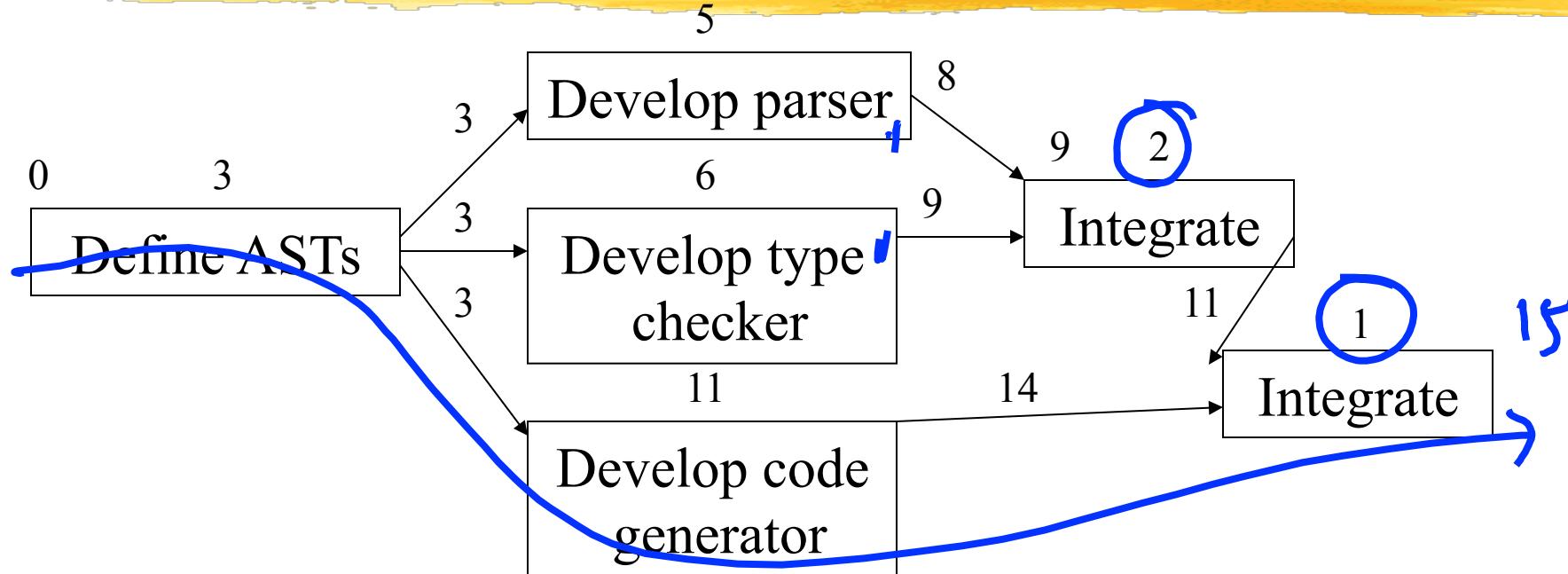
- End = start + duration
- Start = maximum of ends of predecessors

PERT charts (3)



- Time if there are plenty of workers = 17
- Time if there is only one worker = 28
- Time with two workers?

PERT charts (4)



- Critical path - path that takes longest
- Slack - extra time for task
- Only optimize steps on critical path

Project tracking



- Keep track of when each task is finished
 - RUP: When is document done? *REVIEW PASS*
 - XP: When is coding done? *TESTS PASS*
- Compare plan to reality
- Tasks are either finished or not
- Forbid “75% finished” - decompose tasks

When is a task finished?



- Must be objective
 - “Design passed review” not “Design is reusable”
- Different tasks are evaluated differently
 - In RUP, writing a document is a separate task from reviewing it
 - In XP, a programming task is not finished until all unit tests run correctly

Wrong way to plan



- Vision
- Requirements
- Design
- Construction
- Test

RUP way to plan



Project manager responsible for:

- Project plan (developed during inception phase)
- Iteration plan (developed by end of previous iteration)
- Assessing iteration
- Risk list

Main project plan documents



Inception	Elaboration		Construction			Transition
Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7



Business Case

Software Development Plan

Work Breakdown Structure

Business case



Economics of project

- Context (domain, market, scope)
- Technical approach
- Management approach
 - Schedule
 - Objective measures of success
- Financial forecast
 - Cost estimates
 - Revenue estimates

Software development plan



- Detailed plan for building software
- Describes process, **milestones**, environment (tools, resources), change management (requirements, software), assessment
- Updated as more is learned
- Used by developers to plan their work

Work breakdown structure



- Lists
 - Tasks
 - How much time they will take
 - Who will do them
- Used to budget and control costs
- Depends on design of system
- Gets more detailed over time

RUP project milestones



■ CS427, Fall 2006

- Inception - iter 0 Oct 2-Oct 20
- Elaboration - iter 1 Oct 23-Nov 10
- Elaboration - iter 2 Nov 13-Dec 1

■ CS428, Spring 2007

- Construction - iter 3 Jan 22-Feb 16
- Construction - iter 4 Feb 19-Mar 16
- Construction - iter 5 Mar 19-Apr 6
- Construction - iter 6 Apr 9-Apr 27
- Transition - iter 7 Apr 30-May 4

Manpower on RUP project



- Inception - 3 experienced people
- Elaboration - 6 more people
- Construction - 12 more people
- Transition - lose half the group

Scheduling in RUP



- Tasks - planning, analysis, design, programming, testing
- Estimating - people doing the task make the estimates
- Scheduling
 - Top-down schedule based on prior projects
 - Bottom-up schedule based on task-level planning and metrics from previous iteration

Default top-down budget



■ Management	10%
■ Environment	10%
■ Requirements	10%
■ Design	15%
■ Implementation	25%
■ Assessment	25%
■ Deployment	5%

Tracking in RUP



- Major milestones
 - Prototype, Alpha, Beta, Product
- Minor milestones
 - Every iteration
 - Customer can review
- Assessments
 - Developer review, internal tests

Questions on planning



- How do you decide how much work can be done in a particular iteration?
- How do you know how many people it will take to get all the features done in time?
- How do you know who programs a particular class?
- How do you decide what features get done in each iteration?

Next: Guest lecture



- Brian Runk from Morgan Stanley
 - U of I alum (BS in CS, 1998)
- “Patching Production Software”
- Read Chapter 6 of Hamlet and Maybee, “The Test Plan”

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Visitors



- Brian Runk from Morgan Stanley
 - “Patching Production Software”
 - Your comments on guest lecture?
- More visitors this semester
 - ACM hosts 12th annual student conference
 - Oct 20-22
 - <http://www.acm.uiuc.edu/conference/2006/>

Class schedule



- Homework 1 was due today
 - Graded by Oct 6
 - We will not eliminate many proposals
 - You can start forming larger groups now
 - Preferred group size: 8 students
- ~Oct 13: “Project fair”
- Oct 10: Midterm
- Oct 24: Homework 2

Lecture schedule



- Today: Managing Schedules
- Sep 28: Architecture
- Oct 3: UML
- Oct 5: Analysis and Design in RUP
- Oct 10: Midterm

Not a (detailed) schedule



Inception

Elaboration

Construction

Transition

Preliminary It.	Iter. 1	Iter. 2	Iter. 3	Iter. 7	Iter. 8	Iter. 9
-----------------	---------	---------	---------	---------	---------	---------

Example project schedule



- Database for Marching Illini Uniform
- Inception Phase (3 weeks)
 - Form the team.
 - Choose the software engineering process.
 - Choose the team members' roles.
 - Get requirements from customer.
- First Iteration (3 weeks)
 - Decide database storage location.
 - Select database technology.
 - Design database.
 - Implement database.
- Second Iteration (2 weeks)
 - Finalize requirements on interface to the database.
 - Design structure for interface.
 - Implement interface to database.
- Third Iteration (3 weeks)
 - Test and debug software.
 - Design installer for program.

Your example schedule



- Volunteer to have your project discussed
- Remember to update your schedule!

Planning



- Must know scope
 - Requires some analysis/design
 - Based on history
 - Always risky
-
- Remember to update your plan!

Scope



- The customer's needs
 - The business context
 - The project boundaries
 - The customer's motivation
 - The likely paths for change
-
- May modify scope as you form large groups

How to learn scope



- Work with customer
- Ask questions
- Work on document together
 - Define problem
 - Propose solutions
 - Evaluate potential solutions
 - Specify first set of requirements
- Your examples: who talked with customers?

Example Scope (1)



- Build a web site for Journal of Pattern Languages of Programming. Customers are readers, authors, reviewers, editors. Purpose is to create a literature of pattern languages.
- Must register to read or post.
- Readers can see published papers (PDF and HTML), can comment on papers.

Example Scope (2)



- Authors can submit paper and can see status of paper.
- Reviewers can post reviews for papers they are reviewing and can see other reviews.
- Editors can see all status on all papers.

Estimation



- Depends on scope
- Decompose (functional or task)
- Based on history
- Use at least two techniques
- “Estimation” is risky

- Remember to update your estimates!

Task decomposition



- Depends on process
- XP is based on stories
- RUP is based on use cases, architecture, components

RUP task decomposition



- Systems analyst lists use cases
- Use case specifier describes them in detail
- Architect selects the most important
- Use-case engineer makes “realization”
- Component engineer makes analysis classes and packages (groups of classes)
- ... for design, implementation, and testing

RUP estimation



- Measure amount of time needed to analyze, design, implement and test a use case
- Given number of unanalyzed use cases, estimate total work needed to finish
- Given number of analyzed use cases, estimate total work needed to finish
- Given number of designed use cases, ...

Functional decomposition



- Divide program into pieces
- Estimate the cost of each piece
 - Lines of code / speed of developers
 - Function points / speed of developers
 - Man-months (mythical)

Example with journal (1)



Version 1: implement in PHP, 2 people

- Registration - 2 man-weeks
- Readers - 1 man-week
- Authors - 1 man-week
- Reviewers - 3 man-weeks
- Editors - 6 man-weeks

Example with journal (2)



Version 2: reuse OOPSLA system, 2 people

- Learning Java and servlets - 3 weeks
- Changing registration - 1 man-week
- Readers - 1 man-week
- Editors - 2 man-weeks

Scheduling



- Version 1: PHP
 - $13 \text{ man-weeks} / 2 \text{ people} = 6.5 \text{ weeks}$
- Version 2: OOPSLA
 - $3 \text{ weeks learning} + 4 \text{ man-weeks} / 2 \text{ people} = 5 \text{ weeks}$

Example other considerations

- Ann and Bob really want to learn Java
- Management wants to get into Java

Consideration in your project



- Volunteer to have your project discussed

Planning effort - COCOMO

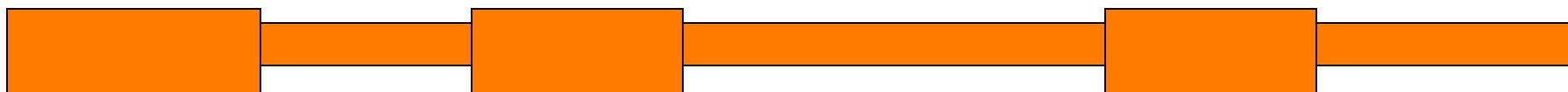


- Given size of system, will predict effort
- Size = KLOC, FP, other
- Effort = c (Size) P
- Coefficient c depends of many factors
 - product complexity
 - personnel experience
 - required reliability
- P is between 1.01 and 1.26

Managing a RUP project



Inception	Elaboration	Construction			Transition
Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6



Planning



Requirements



Architecture

Incremental planning



- Make detailed plan for next iteration
 - Make fuzzy plan for other phases
 - Repeatedly revise plans
-
- RUP has project plan and iteration plan
 - Remember to update your plan!

Scheduling in XP



- Zero' th iteration
 - write stories
 - prototype until developers can estimate
 - create “system metaphor”
- Set of stories, with estimate for each
- Project velocity - how much work for iteration
- Schedule - tasks in each iteration

Tracking in XP



- Iterations are fixed length (1,2,3 weeks)
- Length never varies; work varies
- If developers need less or more work, they talk to customer to decide what to move
- Project velocity determined by measuring amount of work done

Planning XP project



- Developers estimate stories
- Customer uses velocity to place stories in groups for each iteration
- Customer gives one iteration of stories to developers
- Actual number of stories done during iteration becomes the new velocity

Planning XP



- Customer can report long-term schedule to his boss
- Customer can
 - add stories
 - revise, remove, reschedule stories
- Developers only worry about stories in current iteration (except for estimation)

Characteristics of XP planning



- Requires a few iterations to make good estimates
- Next iterations are easier to predict than iterations that are far away
- Easy to change schedule by changing stories

Order



- Build system incrementally
- Which part is built first?
 - If A depends on B, build B first
 - Risky part first
 - Part you know first
 - Most valuable to the customer first

Summary questions



- Is XP planning more incremental than RUP planning?
- How do you estimate in XP?
- How do you estimate in RUP?

Next: Architecture



- Read chapter 11 of Hamlet and Maybee, “Software Design”
- Start reading Fowler
 - Midterm includes chapters 1-9 (105 pages)

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Potential changes



- Midterm exam likely to be in the evening
 - Likely time: 7pm on Oct 10
 - Hard to reserve rooms for class time
 - Can take make up exam at class time
- “Project fair” likely to be moved earlier
 - Depends on how you form groups
 - Wiki will list projects (not just proposals)

Some questions and answers



- When will HW1 be graded?
 - By Oct 6
- When will proposals be eliminated?
 - By Sep 29, start forming 8-student groups
- Group with on- and off-campus students?
 - Possible, but not advisable
- What will be on midterm, any study guide?
 - E.g., <http://brain.cs.uiuc.edu:8080/SEcourse/Exams+Fall+06>

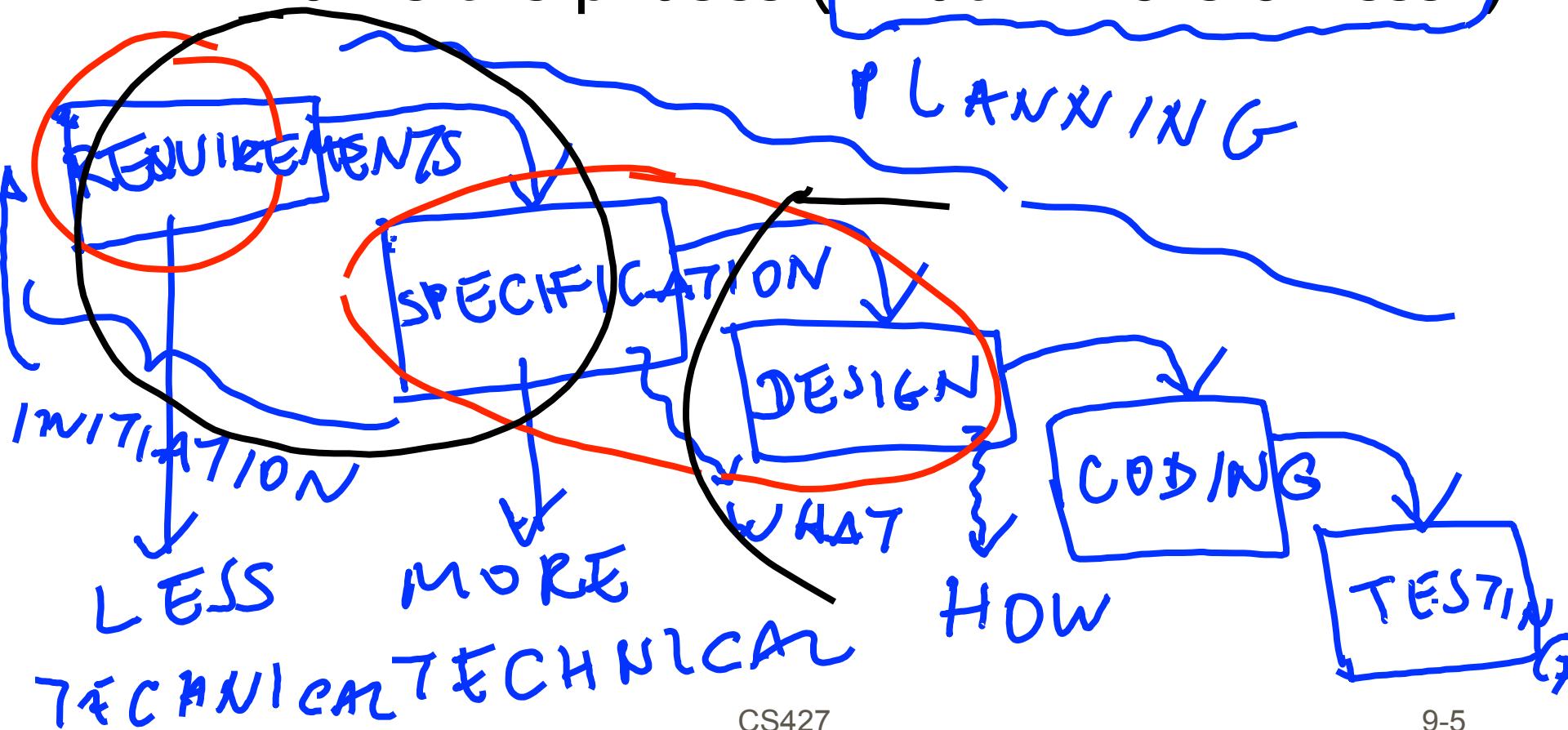
Topics



- Covered
 - Project initiation
 - Project planning
 - Requirements
- Today: Architecture
- Next week: More on design
 - UML: a lot of new, more technical reading

Waterfall model

ME: Name the phases (what if more or less?)



Architecture



- High-level design (components = modules)
- Pervasive patterns (components = lower level objects)
- Early design decisions
 - High-level
 - Pervasive
 - Expensive to change

Importance of architecture



- Helps find requirements
- Key to meeting non-functional requirements (ME: What are those?)
- Divide the system into modules
 - Divide the developers into teams
 - Subcontract work to other groups
 - Find packages to reuse

How to develop architecture?



- Use an architecture you've seen before
 - Invent one
 - Let it emerge
-
- ME: Which process favors which way?

What order?



- Use cases first?
- Processes first?
- Data model first?
- GUI first?

- Depends on architectural style

Architecture in RUP

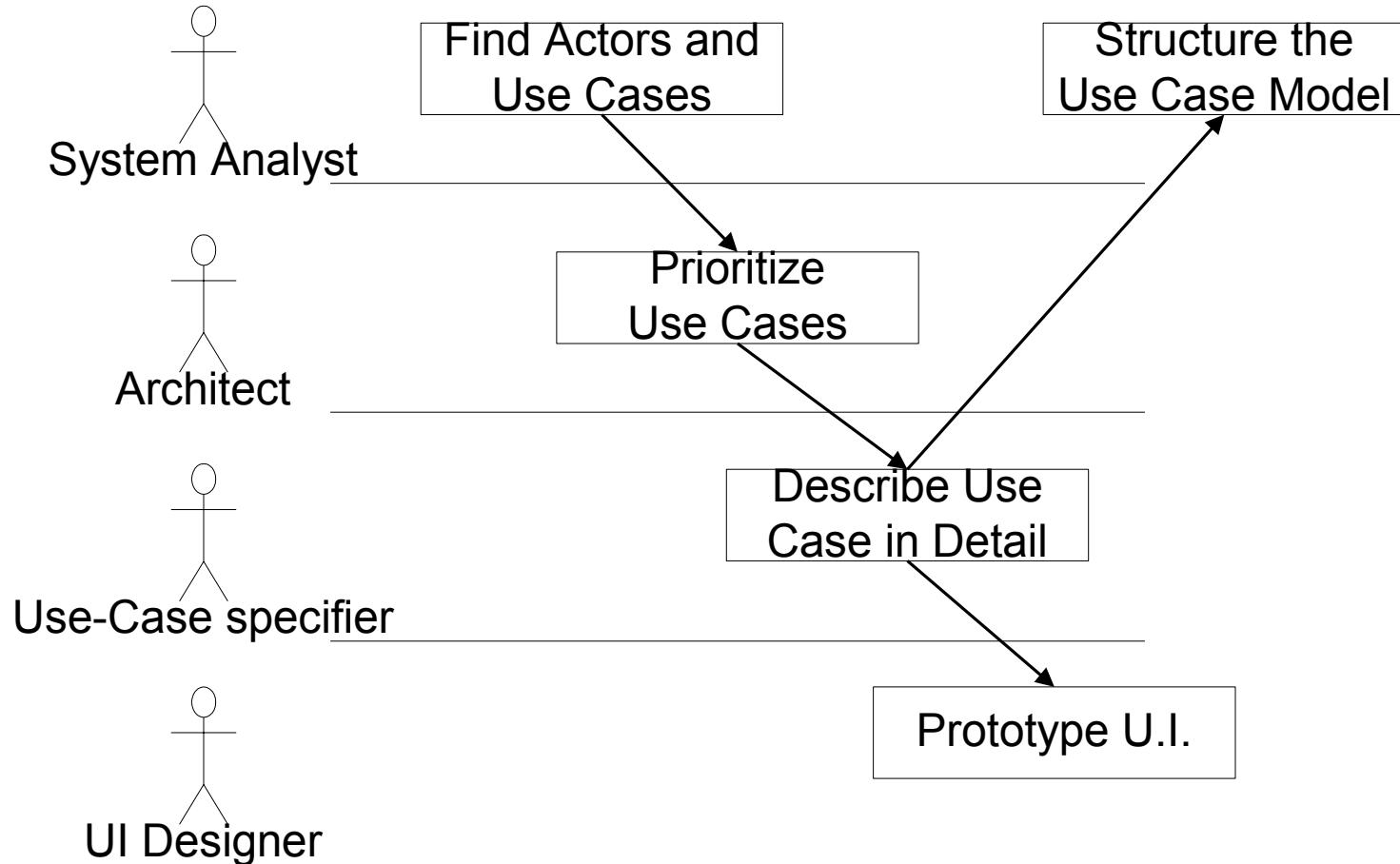
- Use cases
 - Analysis model
 - Design model
- Architecture

ME: Name the phases in RUP

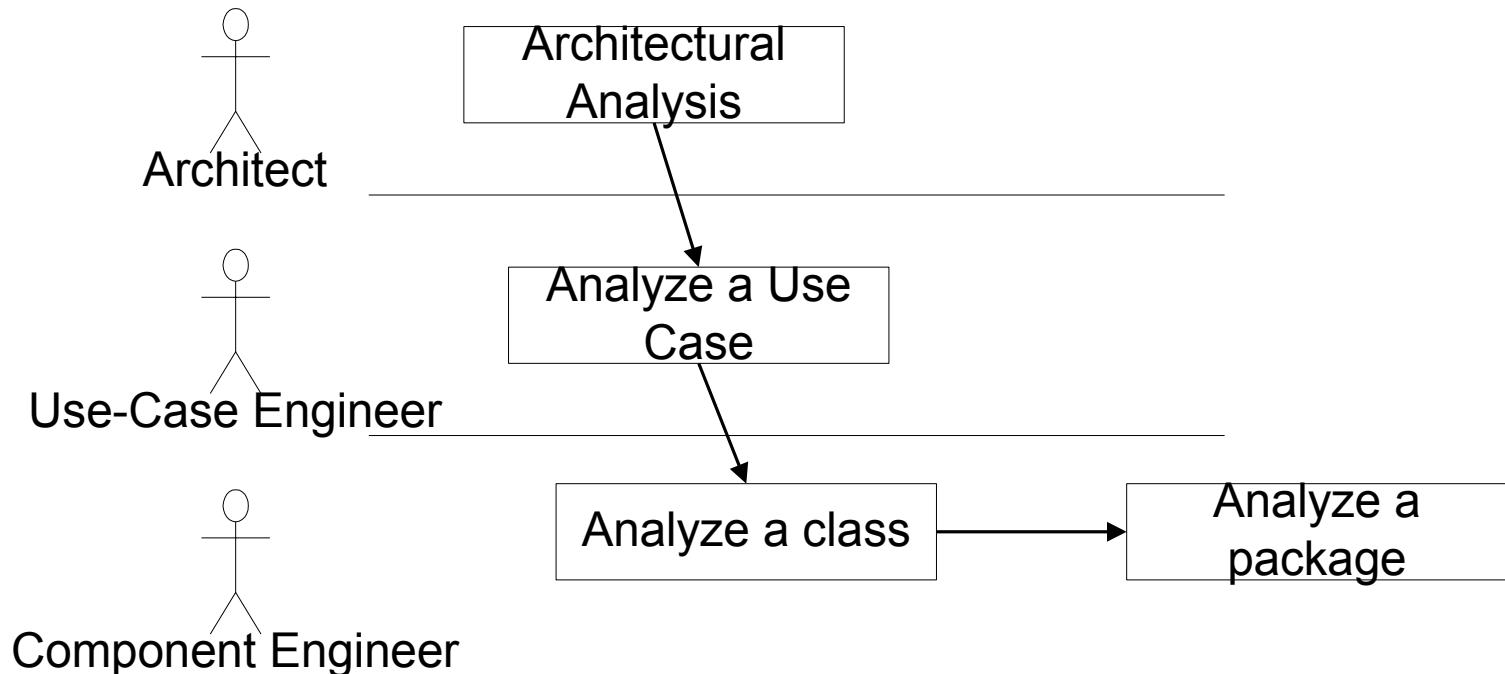
INCEPTION (INITIATION) | ELABORATION | CONSTRUCTION | TRANSITION

Preliminary It.	Iter. 1	Iter. 2	Iter. 3	Iter. 4	Iter. 5	Iter. 6
-----------------	---------	---------	---------	---------	---------	---------

Workflow for capturing requirements



Workflow for analysis



Architecture in XP



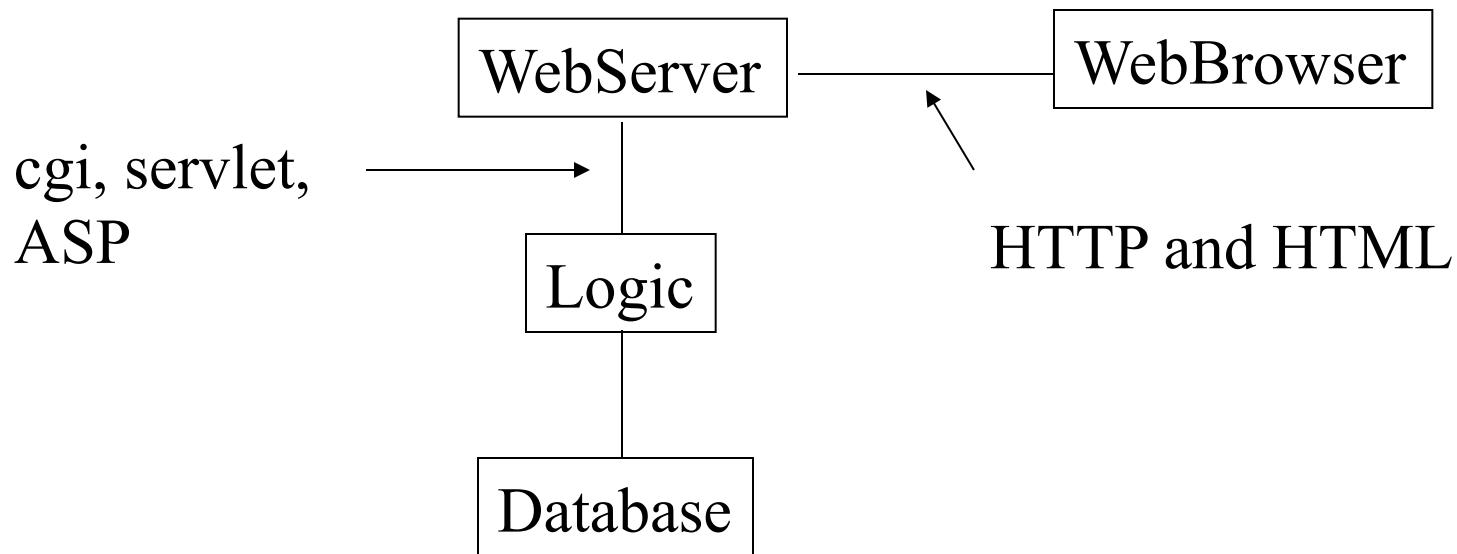
- No architect
- “System metaphor” is the architecture
- Infrastructure emerges, not planned in advance
- Good developers will convince customer to pick important user stories first

Picking an architecture



- Many standard architectures
- Each has strengths and weaknesses
- Each solves some problems and creates others

Architecture of Amazon.com

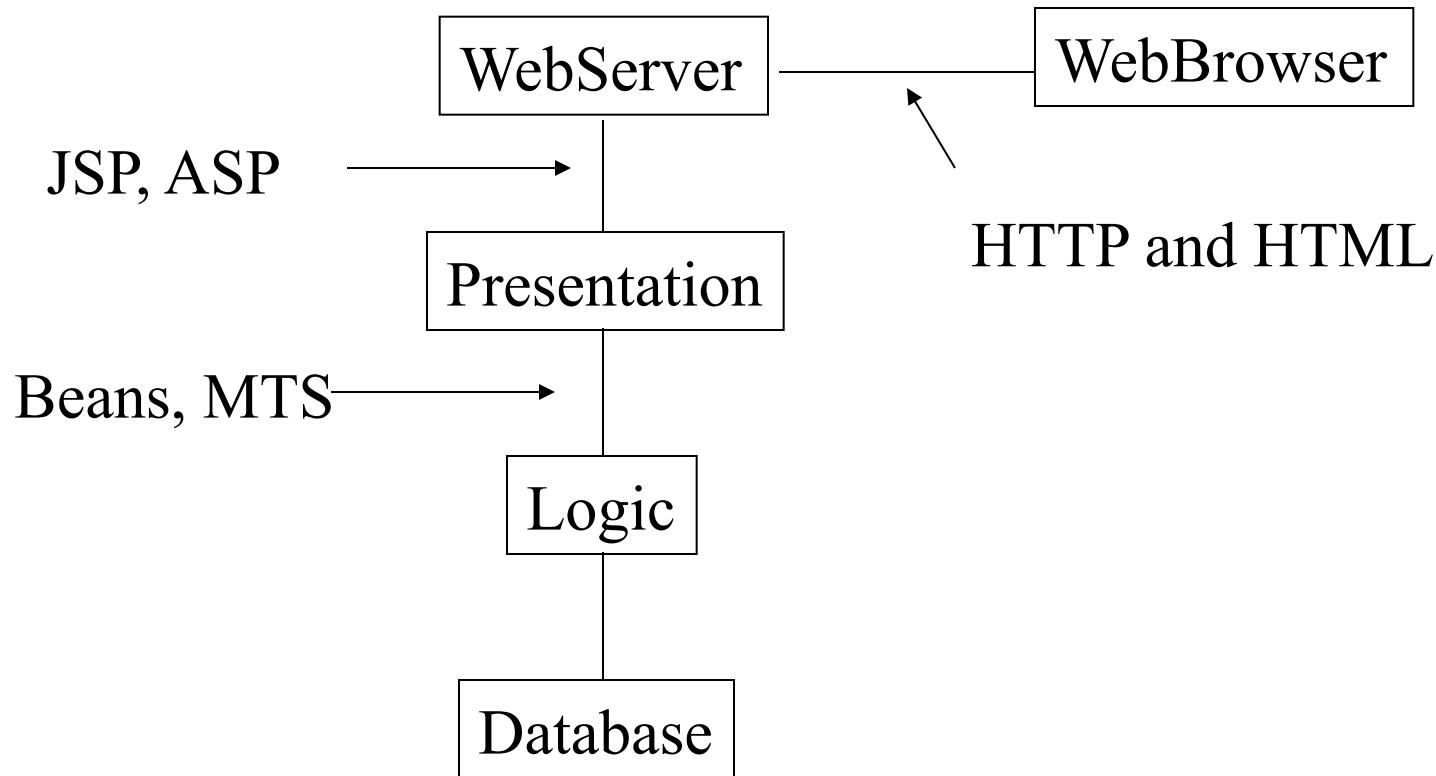


Facts about Web architecture

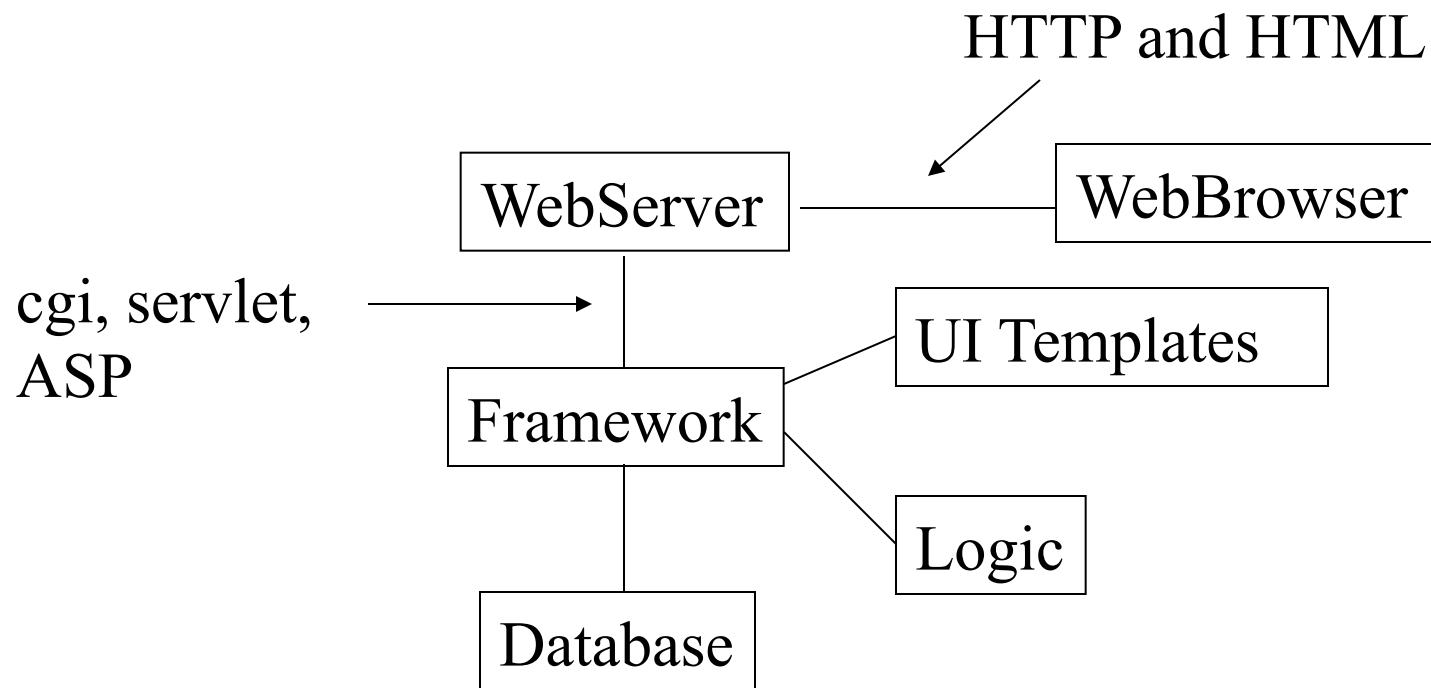


- HTTP protocol is stateless
 - state encoded in URLs and cookies
- Web server should be stateless
 - All state is in database
- Separate “presentation” from logic and database design

One solution



Another solution



Organizing project: people



- Database designer
- UI designer
- Programmers

Organizing project: process



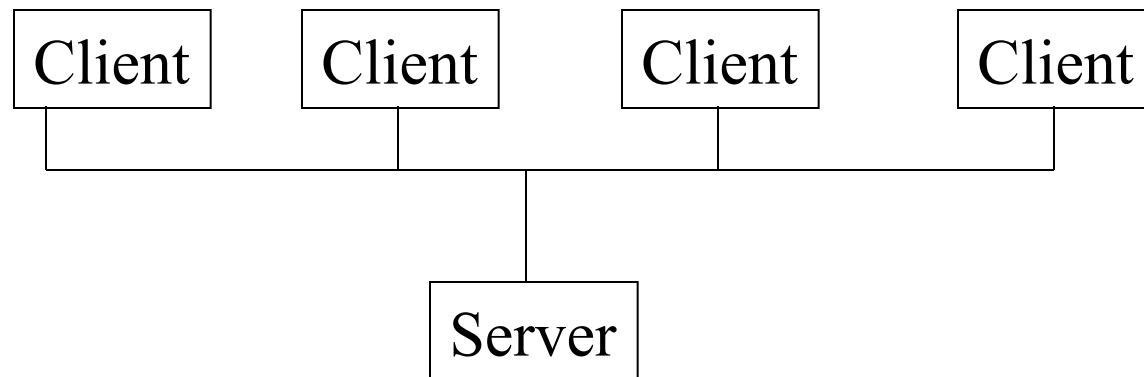
- Database design is key
 - Hard to change
 - Everything else depends on it
- GUI tells what data is needed
 - Build prototype early
 - Redo at the end after feedback
 - Relatively easy to change

Second example architecture



- Online card game
- Java client
- Server used for finding people to form a game
- Once game starts, server not used

Card game architecture



Card game architecture



- Use server only to find other clients, but game is peer to peer
 - More scalable
 - Can have problems with firewalls
 - Harder to make secure
- All communications goes through server
 - Not as scalable
 - More reliable, secure

Card game architecture



- Peer to peer
 - Random numbers, logic is in client
 - Each client broadcasts events that it generates
 - Usually only one client is able to generate events at any one time
- Server
 - Random numbers, logic is in server

Your example architecture



- What two options you may consider?

Trade-offs



- How do you pick an architectural style?
- How do you make design choices?
- What is important?
 - Flexibility and ease of change
 - Efficiency
 - Reliability

Quality attributes



- Desired properties of entire system, not particular features
- Non-functional requirements
- Usability, maintainability, flexibility, understandability, reliability, efficiency, ...

ATAM



- Architectural Trade-off Analysis Method
 - Collect scenarios (architectural use cases)
 - Collect requirements, constraints, and environment descriptions
 - Describe candidate architectural styles
 - Evaluate quality attributes
 - Identify sensitivity of quality attributes
 - Critique candidate architectures

Trade-offs



- How it is really done
- Experts pick a way that seems best and start to use it
- If problems arise, they reconsider
- Want more reading?
 - General reading on making (big) decisions
<http://www.fastcompany.com/online/38/klein.html>

What is an architect?



- Responsible for choosing architecture
- Responsible for communicating with client
- Responsible for making trade-offs about features
- Chief builder - “Architect also implements”
- Leader of developers

Why architecture?



- A common vision of the design that enables people to talk to each other
- The key design decisions of the system
- Simplified version of design, ideal for teaching new developers

What they don't tell you?



- Good architecture requires experience
- There is more to being an architect than picking the architecture
 - “Chief builder”
 - Enforce conceptual integrity

Next: UML introduction



- Read chapters 3 and 4 of “UML Distilled” by Fowler (third edition)
 - How many of you have second edition?
 - The material was in chapters 4 and 5

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- HW1 graded
 - Projects <http://brain.cs.uiuc.edu:8080/SEcourse/Projects+Fall+06>
 - Keep forming 8-student groups
 - Current group members decide on new members
- Midterm: confirmed time and room change
 - 7pm on Oct 10 (Tue) in 1404 SC (not DCL!)
 - Makeup exam: 12:30pm on Oct 10 in 1310 DCL
 - Let us know if you have other conflicts
 - Samples <http://brain.cs.uiuc.edu:8080/SEcourse/Exams+Fall+06>

Topics



■ Covered

- Project initiation and planning
- Requirements (use cases)
- Design (architecture)

■ Today: UML introduction

- Notations useful for requirements, specification, design, and (potentially) coding

Modeling notations (1)



- Used for both requirements analysis and for specification and design
- Useful for technical people
- Provide a high-level view
- Require training
- Many notations
 - Each good for something
 - None good for everything

Modeling notations (2)



- Help developers communicate
- Provide documentation
- Help find errors (tools check for consistency)
- Generate code (with tools)

Unified Modeling Language (UML)



- Graphical modeling notations for describing and designing (OO software) systems
- 13 kinds of diagrams (in UML 2.0)
 - Structure
 - Class diagrams (today)
 - Behavior
 - Interaction
 - Sequence diagrams (today)

UML class diagrams



- Formed by merging Booch Diagrams and OMT Object Diagrams
- Descendent of Entity-Relationship Diagrams
- Describes data and operations

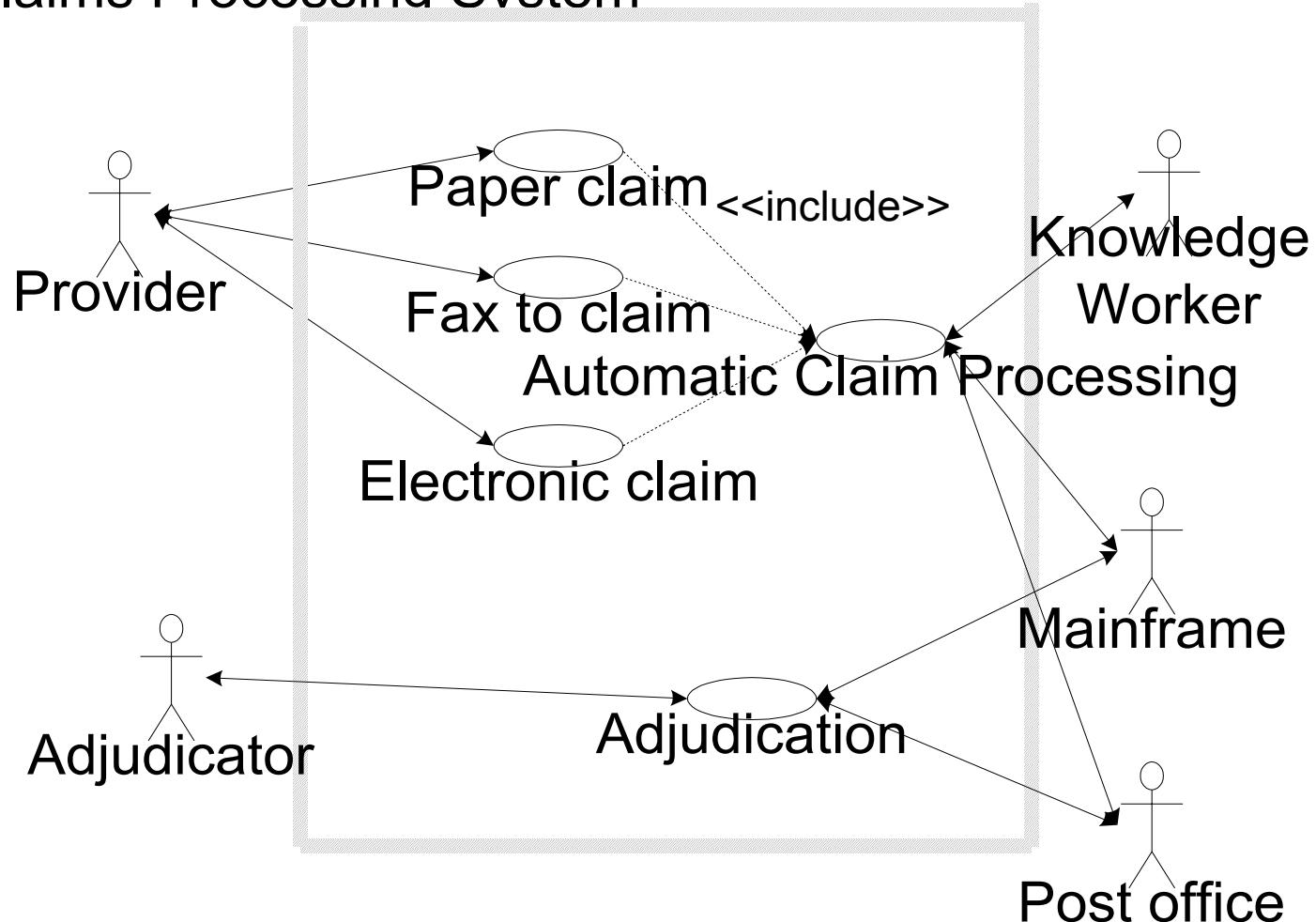
Elements of UML class diagram



- Class
 - Attributes
 - Operations
- Associations
 - Multiplicity
 - Direction/aggregation
- Generalization

Familiar(?) example domain

Claims Processing System

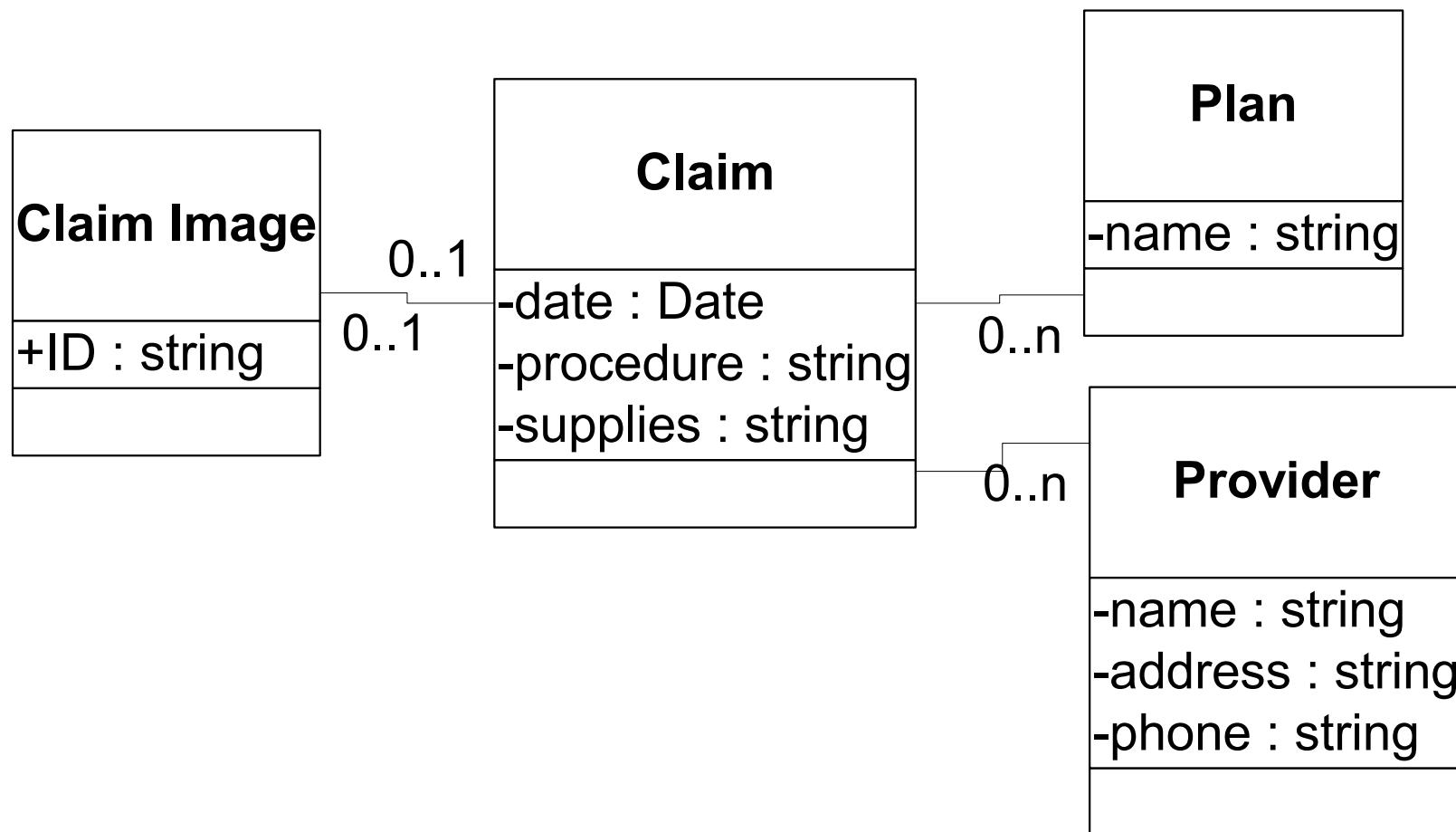


Problem source

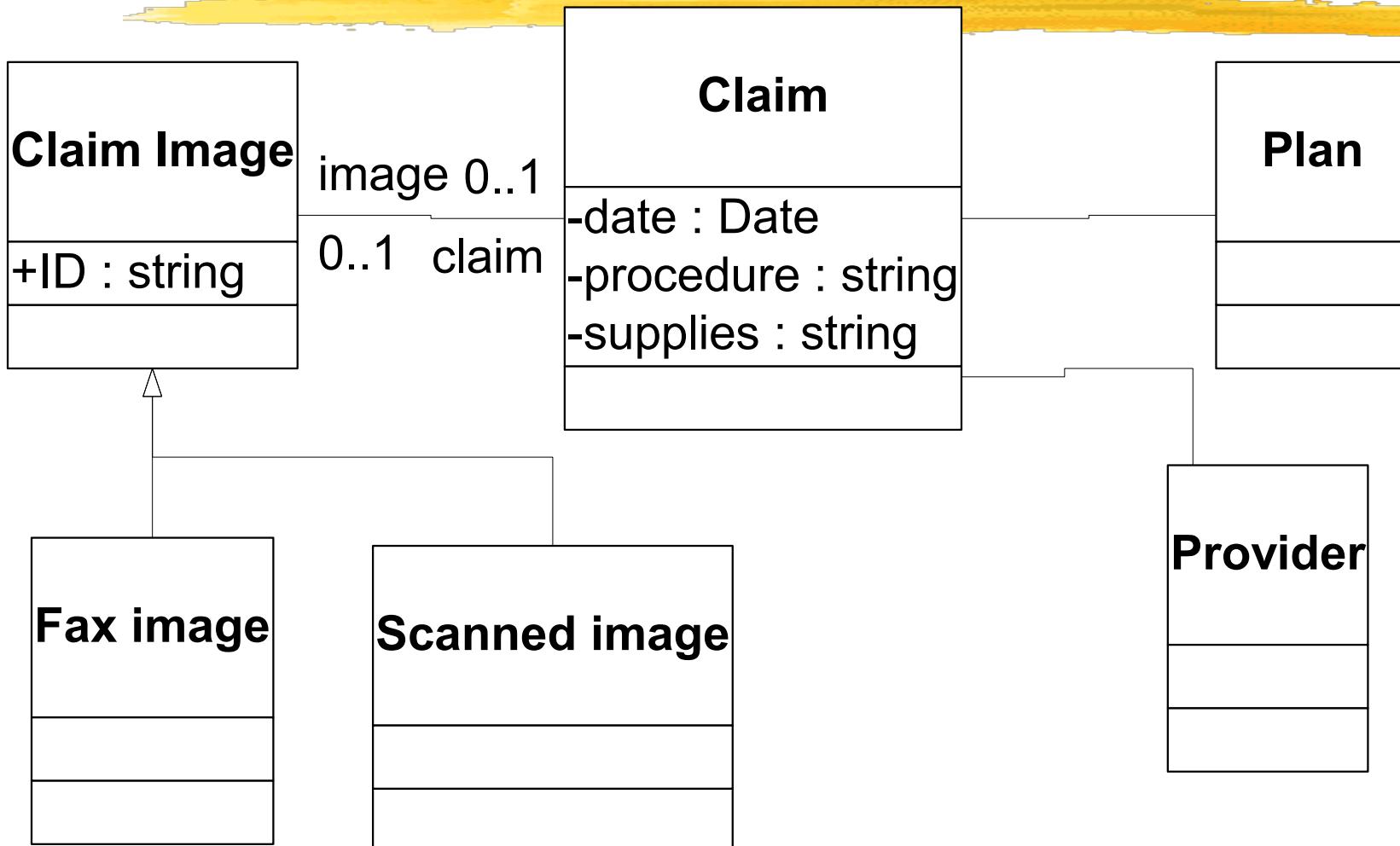


- From a Design Fest at OOPSLA
http://designfest.acm.org/Problems/HealthClaims/HealthClaims_96.htm
- In addition to system description, has three use cases
- Central concept: claim

Claim class diagram



Inheritance



Multiplicity



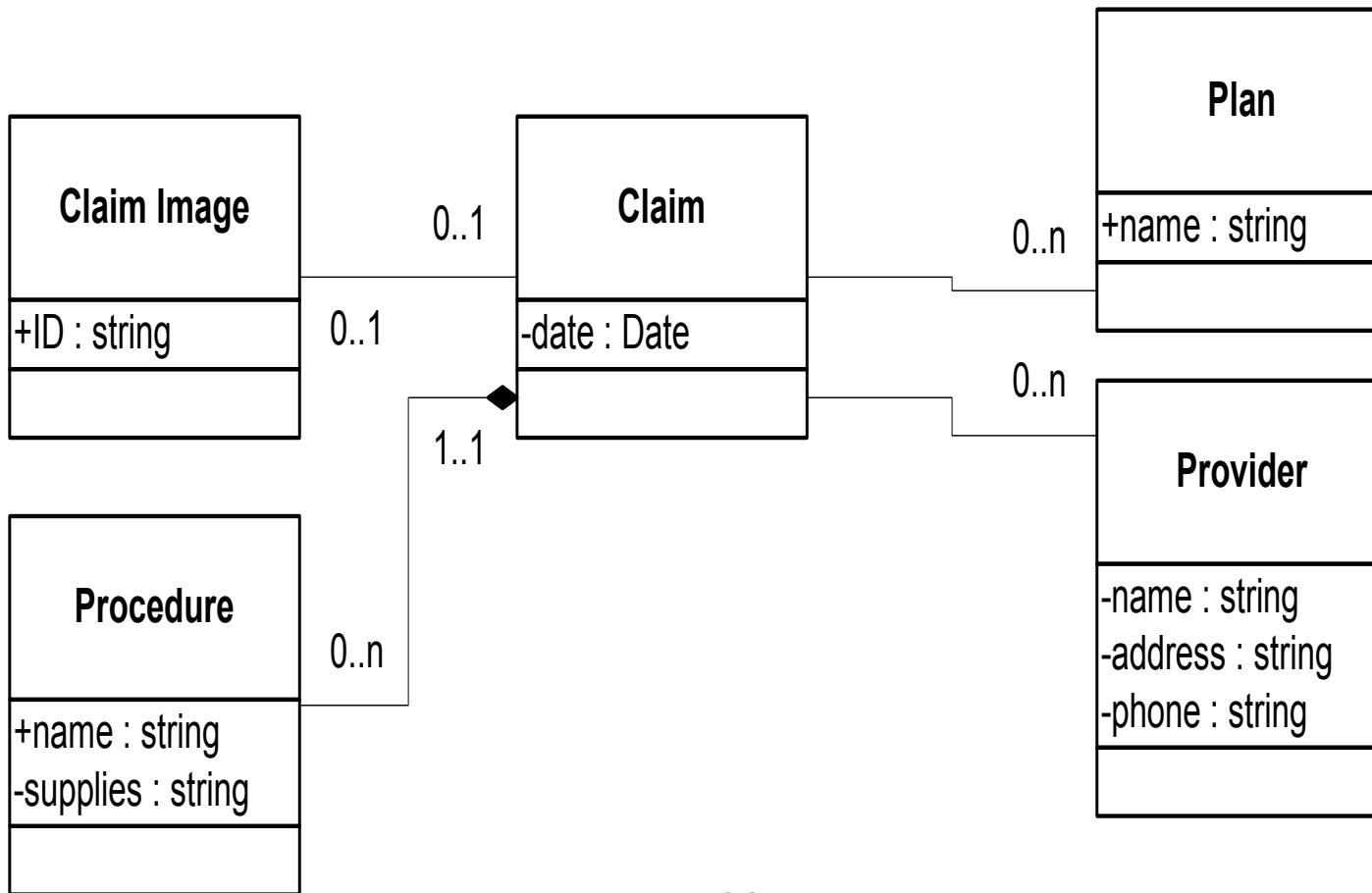
- 0..1 or 0..N is optional
- 1 is mandatory
- * is 0..N

OO modeling points



- Best objects correspond to real-world entities
- Some correspond to actors (adjudicator); they are actually *interfaces* to the actors
- Some correspond to complex processes, but this is an exception; should not be common

Attributes vs. associations

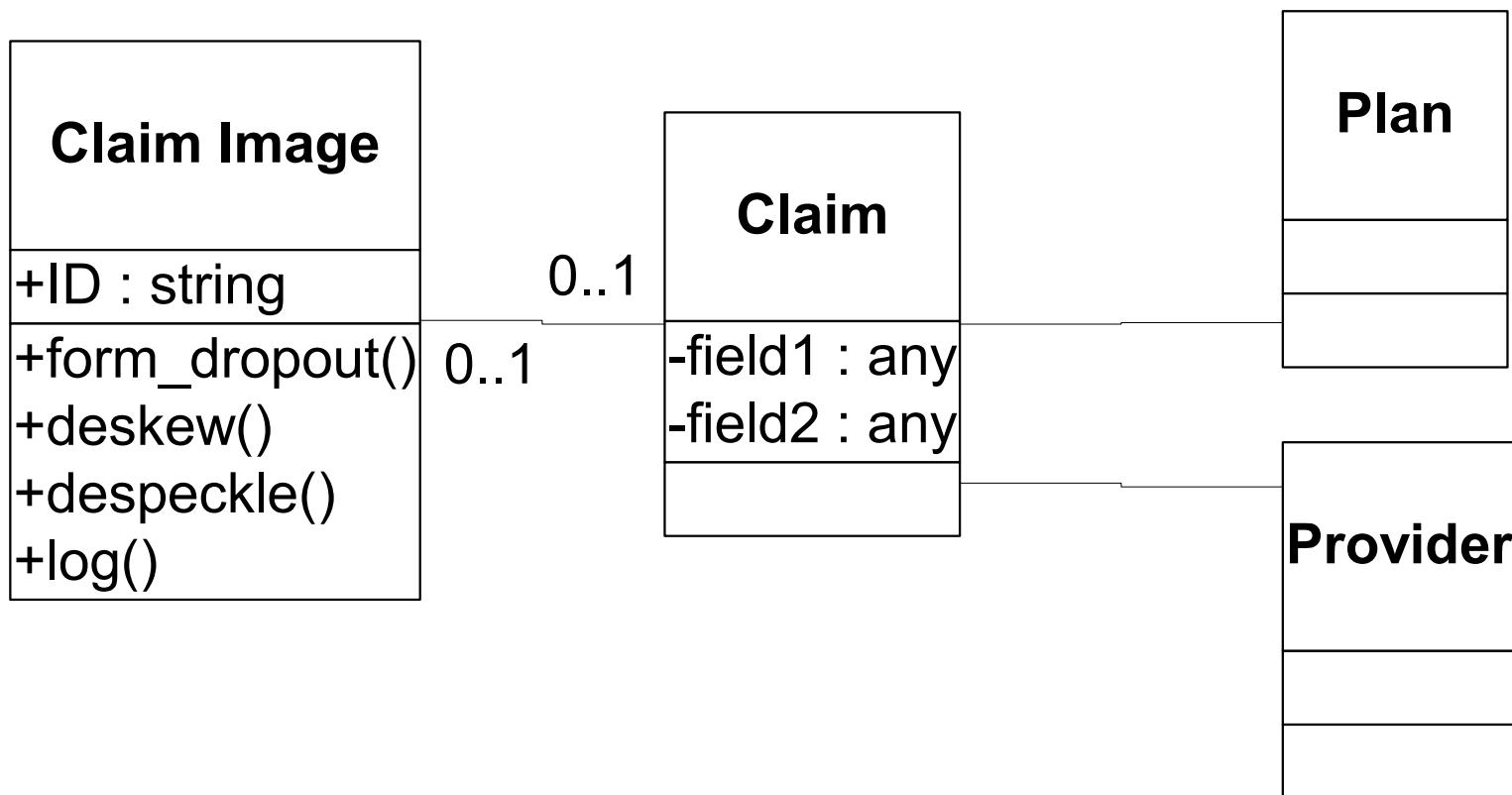


Health claims processing



- Receives health claims and supporting documents via many sources: electronically, fax, on paper.
- Scanned paper and fax processed by OCR. Documents first subject to form dropout, deskewing, despeckling.
- All images are logged to optical disk.

Behavior



OO modeling points



- Class names should be nouns
- Verbs become operations
- Avoid class names ending in “er”

Analysis vs. design



- Class diagrams are used in both analysis and design
- Analysis - conceptual
 - Model problem, not software solution
 - Can include actors outside system
- Design - specification
 - Tells how the system should act
- Design - implementation
 - Actual classes of implementation

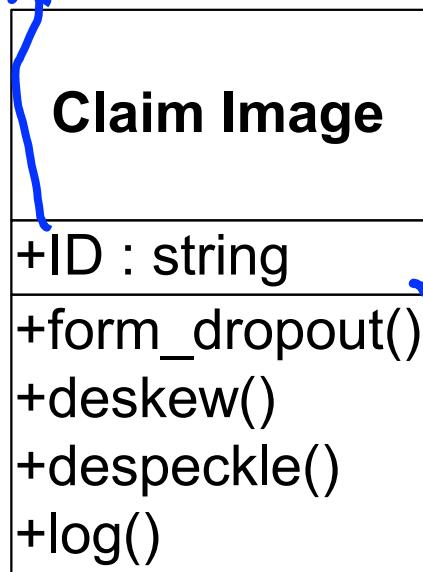
Extending class diagrams



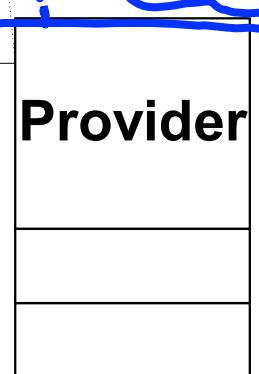
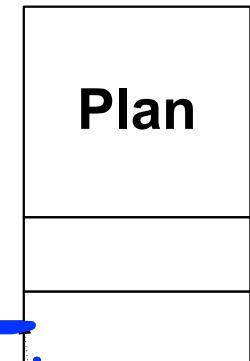
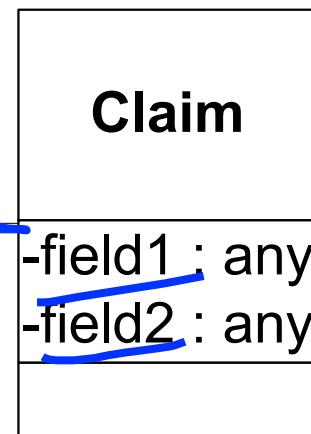
- No modeling notation can do everything
- Modeling notations should be extensible
- UML has two techniques
 - Constraints
 - Stereotypes

Constraint

{ID has at least six

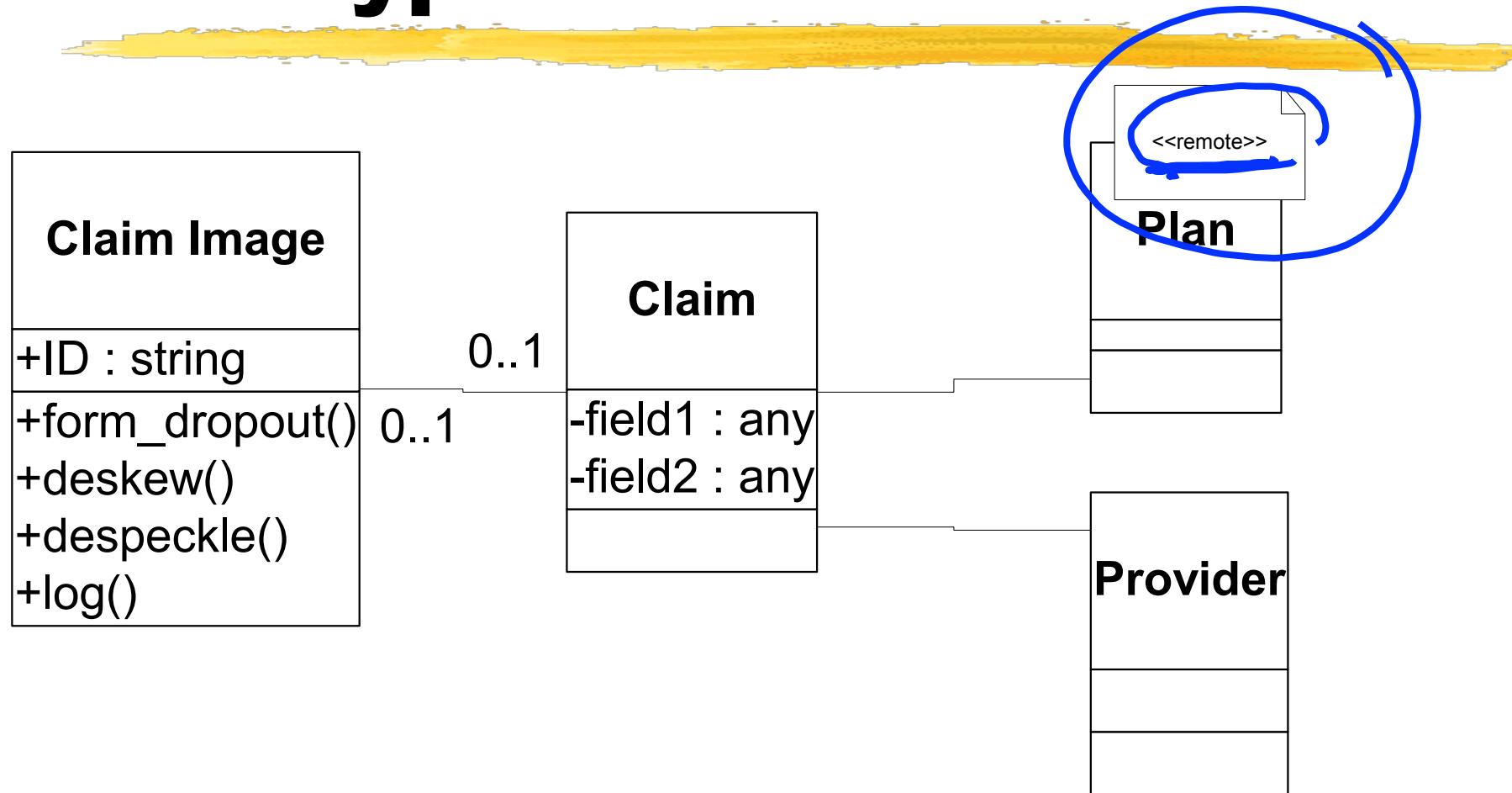


0..1

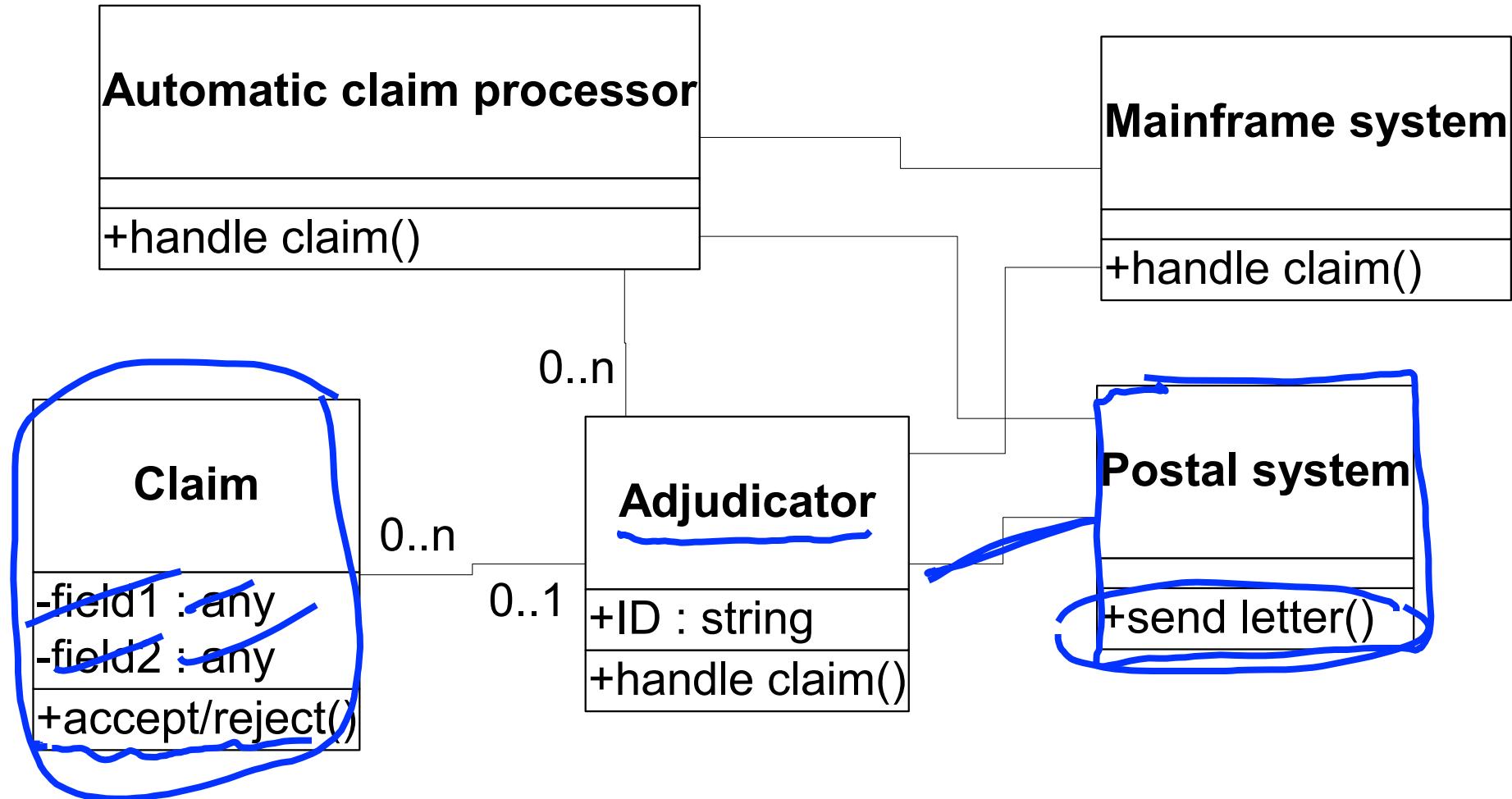


{Provider is legal under Plan}

Stereotype



Interfaces to other systems



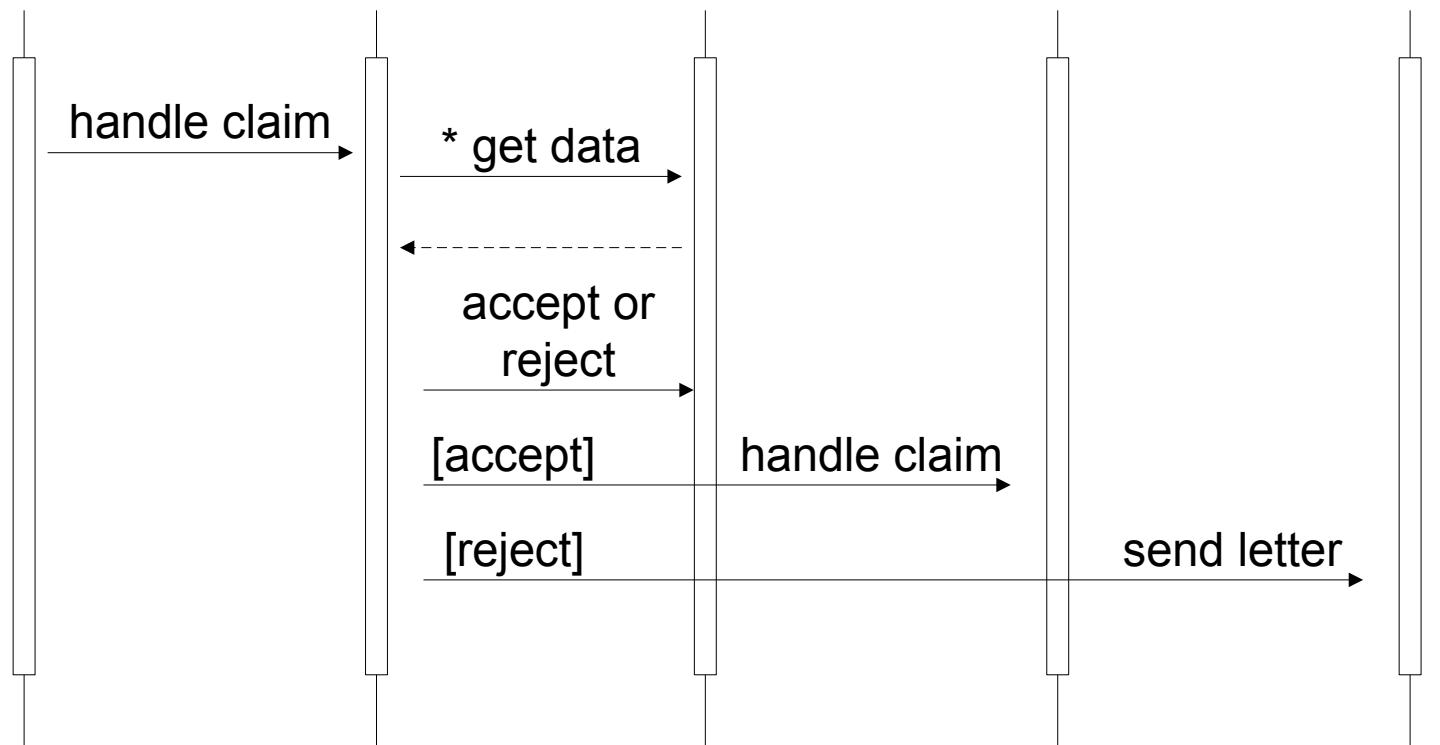
UML sequence diagrams



- Model how a set of objects communicate
- Describe sequence of events (traces)
- A line for each object
- Time goes from top to bottom
- Arrows represent communication events

Sequence diagram for claim

automatic claim proc. an Adjud. a claim mainframe system postal system



Class diagram



- Central model for OO systems
- Describes data and behavior
- In UML, used along with Use Cases and Packages for analysis
- Also used to describe implementation
- Don't confuse analysis and implementation!
 - Separate “what” from “how”

Next: Analysis and design in RUP



- Read chapters 5-8 of “UML Distilled: Third Edition” (“Class Diagrams: Advanced” , “Object Diagrams” , “Package Diagrams” , “Deployment Diagrams” ; 2nd edition: chapters 6-7 and first part of 10)
- Chapters 3-4 listed for today
- Chapters 1-2 and 9 listed earlier

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



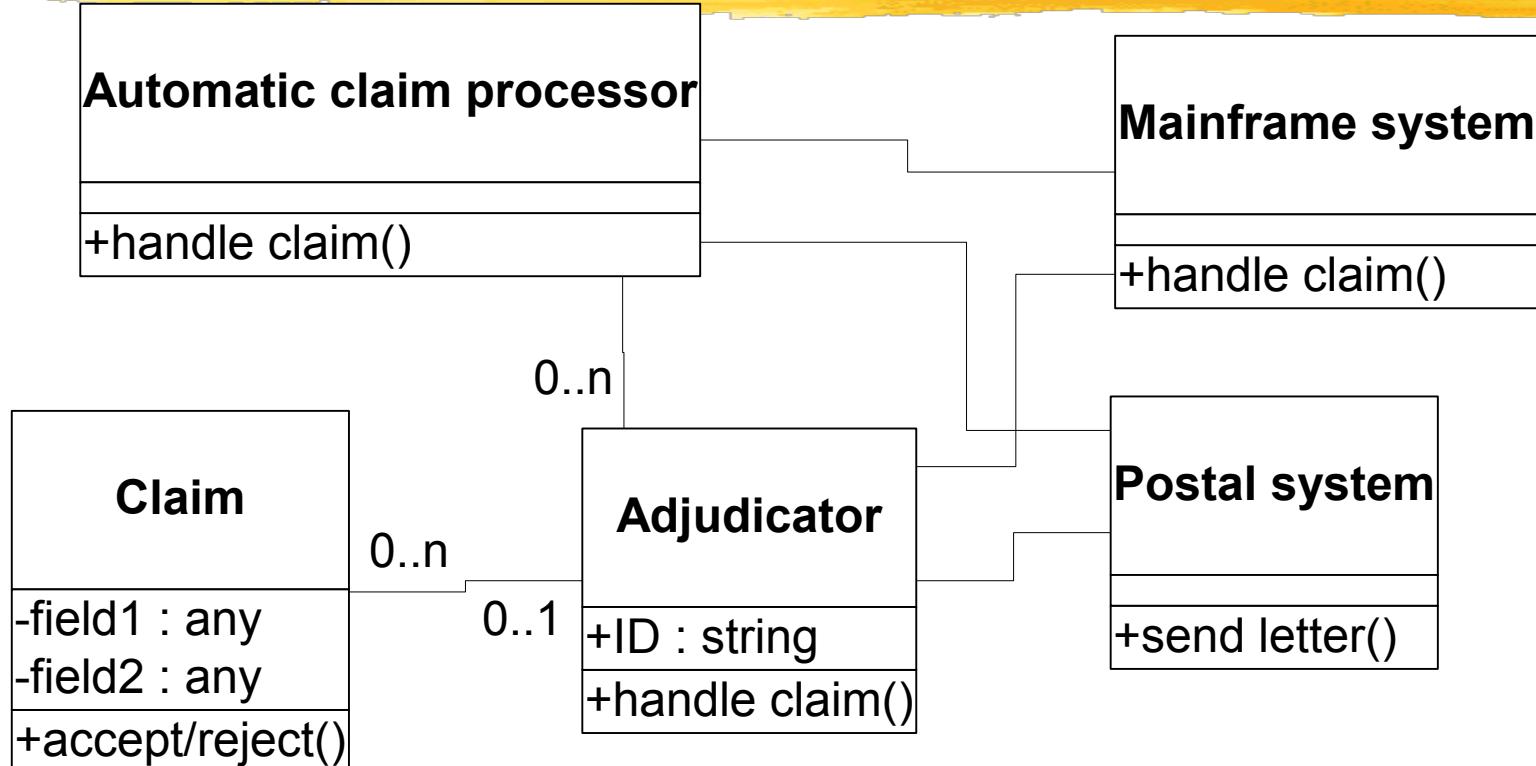
- HW1: Ask for regrading if you think we erred
- Project groups forming slowly
 - Project fair: Oct 6 or Oct 13?
 - Oct 13: Deadline for 8-student groups
- Midterm reminder
 - 7pm on Oct 10 (Tue) in 1404 SC (not DCL!)
 - Makeup exam: 12:30pm on Oct 10 in 1310 DCL
 - Let us know if you have other conflicts
 - “Study guide” and samples available on Wiki

Topics this week



- Unified Modeling Language (UML) intro
 - Graphical modeling notations for describing and designing (OO software) systems, 13 diagrams
 - Structure
 - Class diagrams (previous lecture)
 - Object diagrams (today)
 - Package diagrams (today)
 - Behavior
 - Use-case diagrams (previous lectures)
 - Sequence diagrams (today)
 - Analysis and design in RUP

Reminder: Example class diagram



Where do multiplicities go? Correct here?

Questions? Come to office hours (Thu 2-3, Mon 3-4) or email us!

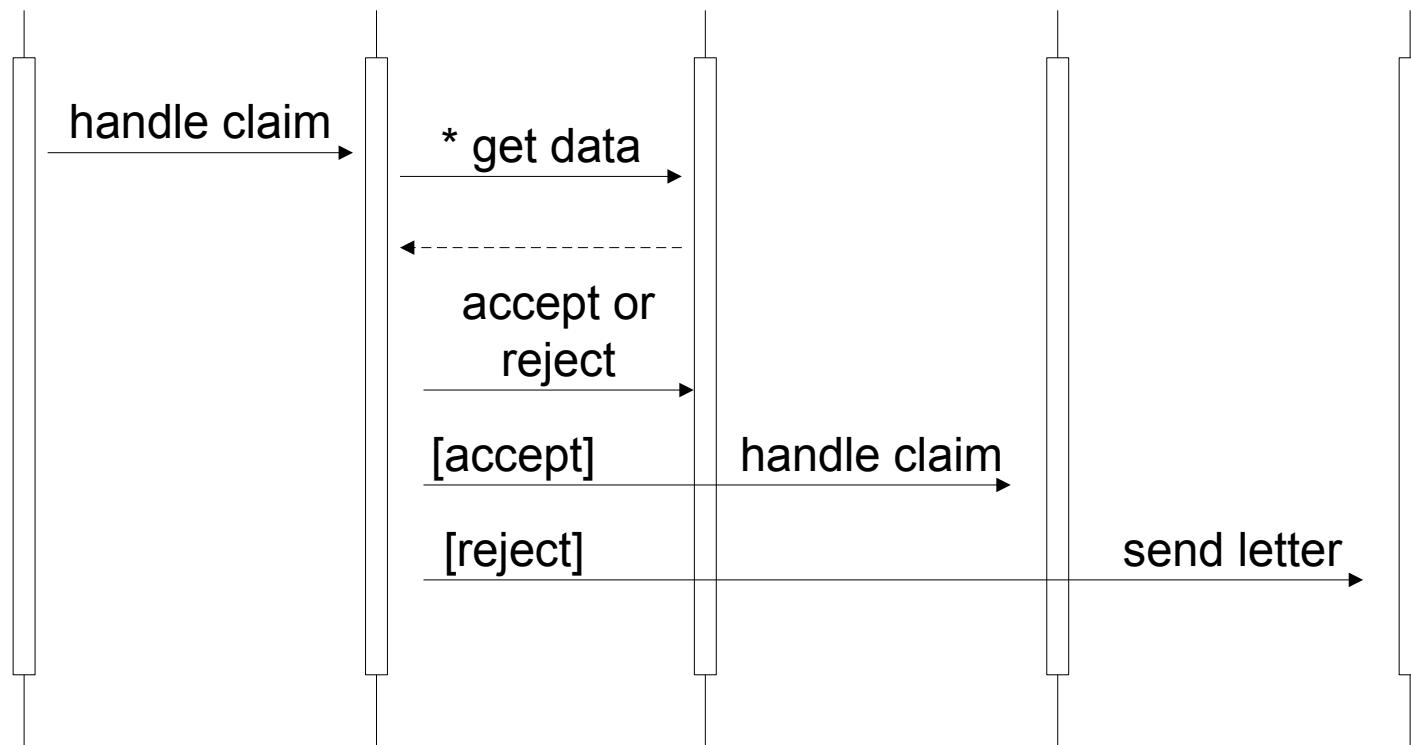
UML sequence diagrams



- Model how a set of objects communicate
- Describe sequence of events (traces)
- A line for each object
- Time goes from top to bottom
- Arrows represent communication events

Sequence diagram for claim

automatic claim proc. an Adjud. a claim mainframe system postal system



Correct lifetime and activation?

CS427

11-6

Summary of class diagram



- Central model for OO systems
- Describes data and behavior
- In UML, used along with Use Cases and Packages for analysis
- Also used to describe implementation
- Don't confuse analysis and implementation!
 - Separate “what” from “how”

Activities in RUP



- Requirements capture: learning what the system is to do
- Analysis: transform requirements closer to the software design; classes and subsystems
 - Ensure that functional requirements are met
- Design: move closer to implementation
 - Ensure that non-functional requirements are met

Result of requirements capture



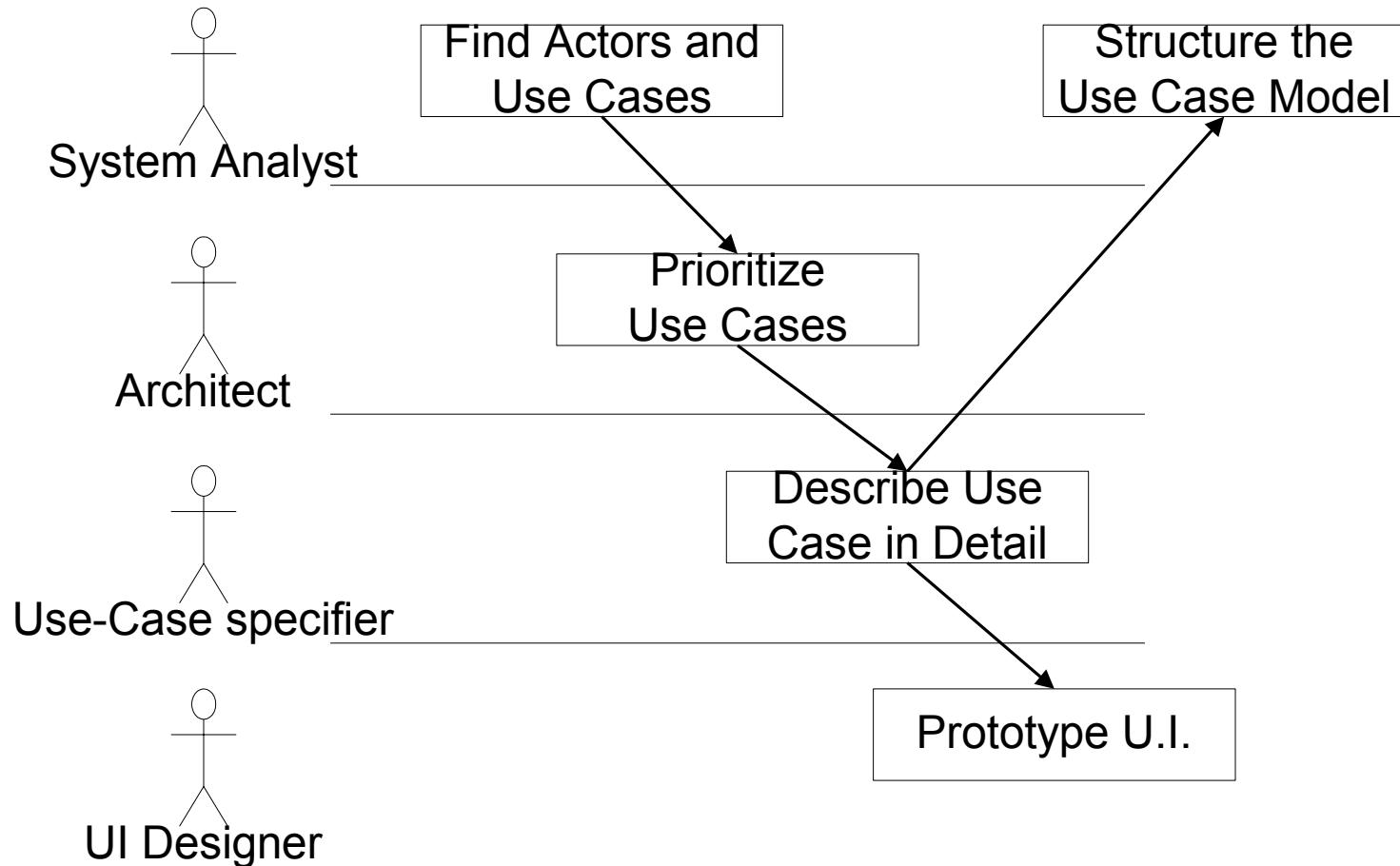
- Vision
- Use-case model with detailed description of each use case
- Set of user interface sketches and prototypes
- Requirements document for generic requirements

Workers and their artifacts



- System analyst
 - Use-case model (diagram)
 - Actors
 - Glossary
- Architect - Architecture description
- Use-case specifier - Use-cases
- User interface designer
 - User interface prototype

Workflow for requirements



System analyst



- Determines actors and use cases based on knowledge of system
- Interviews users to discover more use cases and actors

Architect



- Determine which use cases need to be developed first
- High priority use cases
 - Describe important and critical functionality
 - Security
 - Database
 - Hard to retrofit later

Use-case specifier



- Create detailed descriptions of use cases
- Works closely with real users of use case

UI design



■ Logical design

- Which user-interface elements are needed for each use case?
- What information does the actor need to receive from or give to the system?

■ Prototyping

- Often is on paper
- Test on real users

Requirements specification



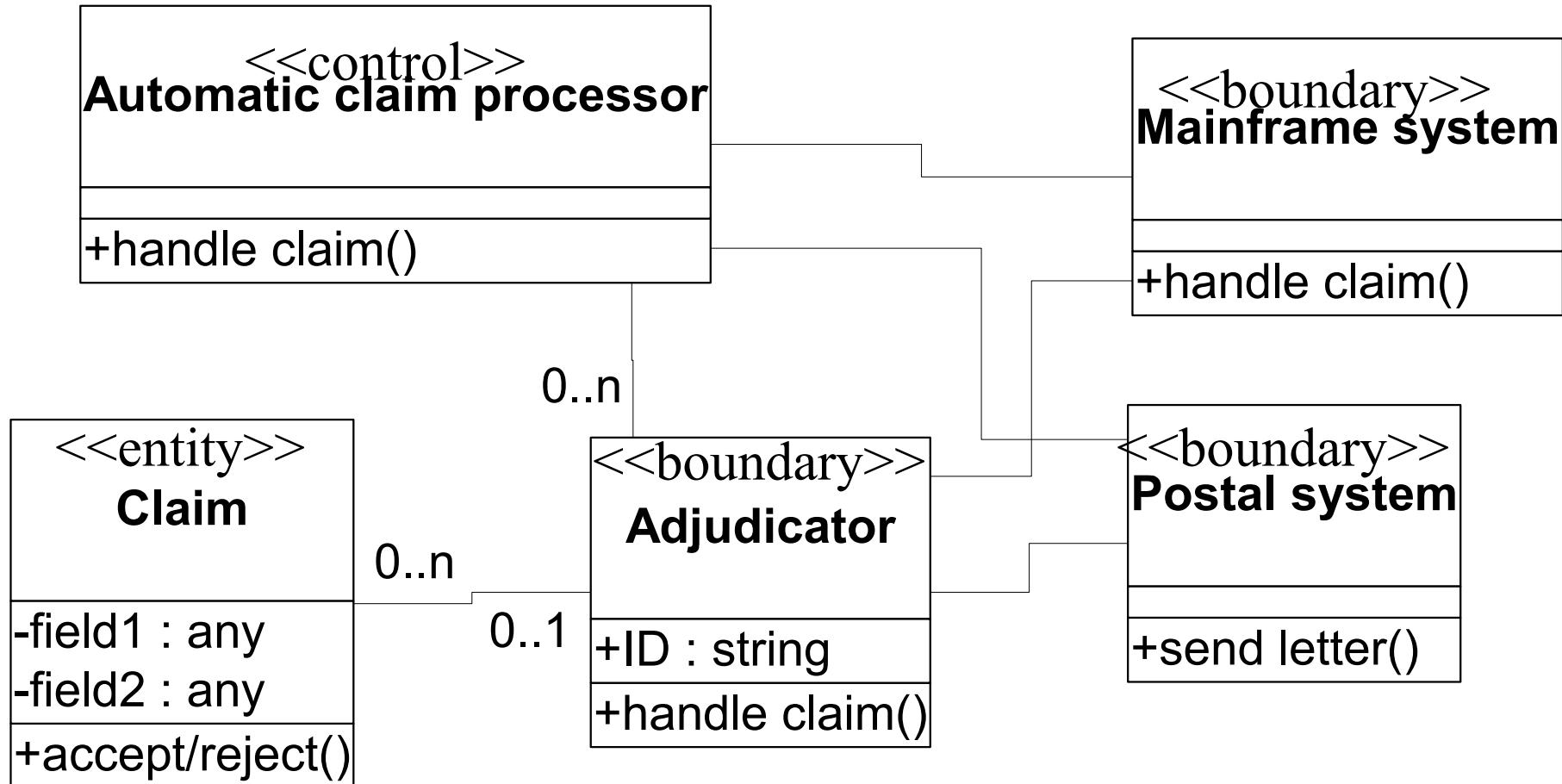
- Not all requirements go in a use case
 - Example: security
 - Example: global performance
- Requirements document describes all other requirements that are not suitable for use cases

Analysis model



- Class diagrams
 - Vague interfaces (“responsibilities”)
 - Vague associations (ignore navigability)
 - Stereotype classes:
 - Boundary - UI, associated with actor
 - Control - control associated with a use case
 - Entity - persistent, the “real” objects
- Use-case realization (Analysis)

Stereotypes

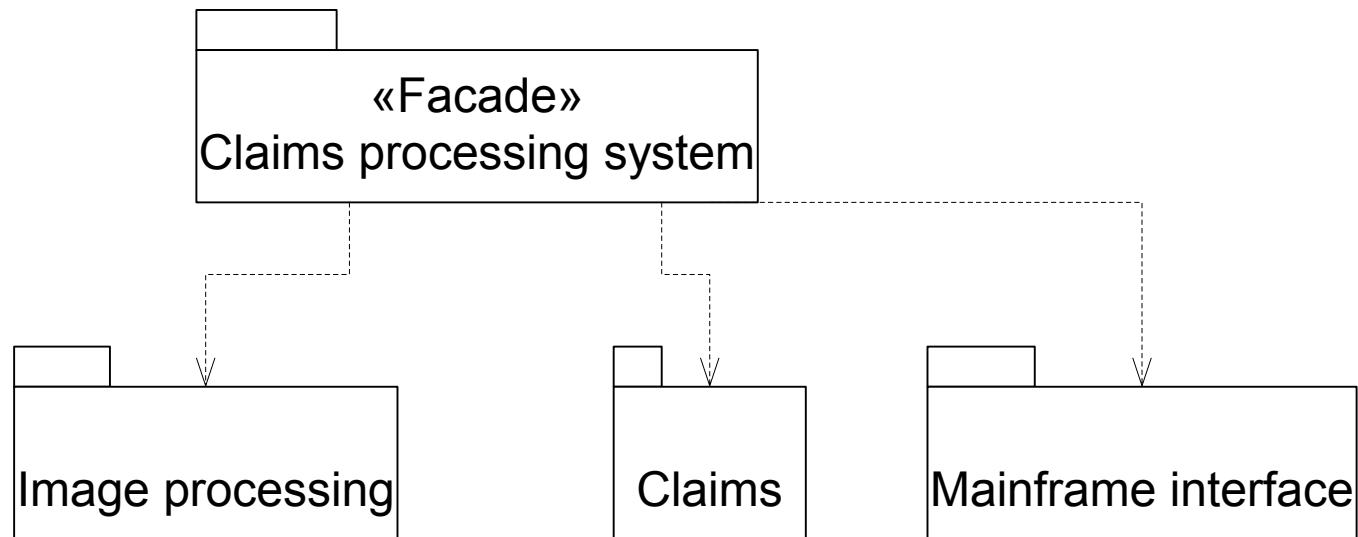


Packages



- Logical grouping
- Used to
 - Divide large system into smaller subsystems
 - Show dependencies between subsystems
- Can contain
 - Class diagrams or packages
 - Use cases, sequence diagrams, etc.

Packages

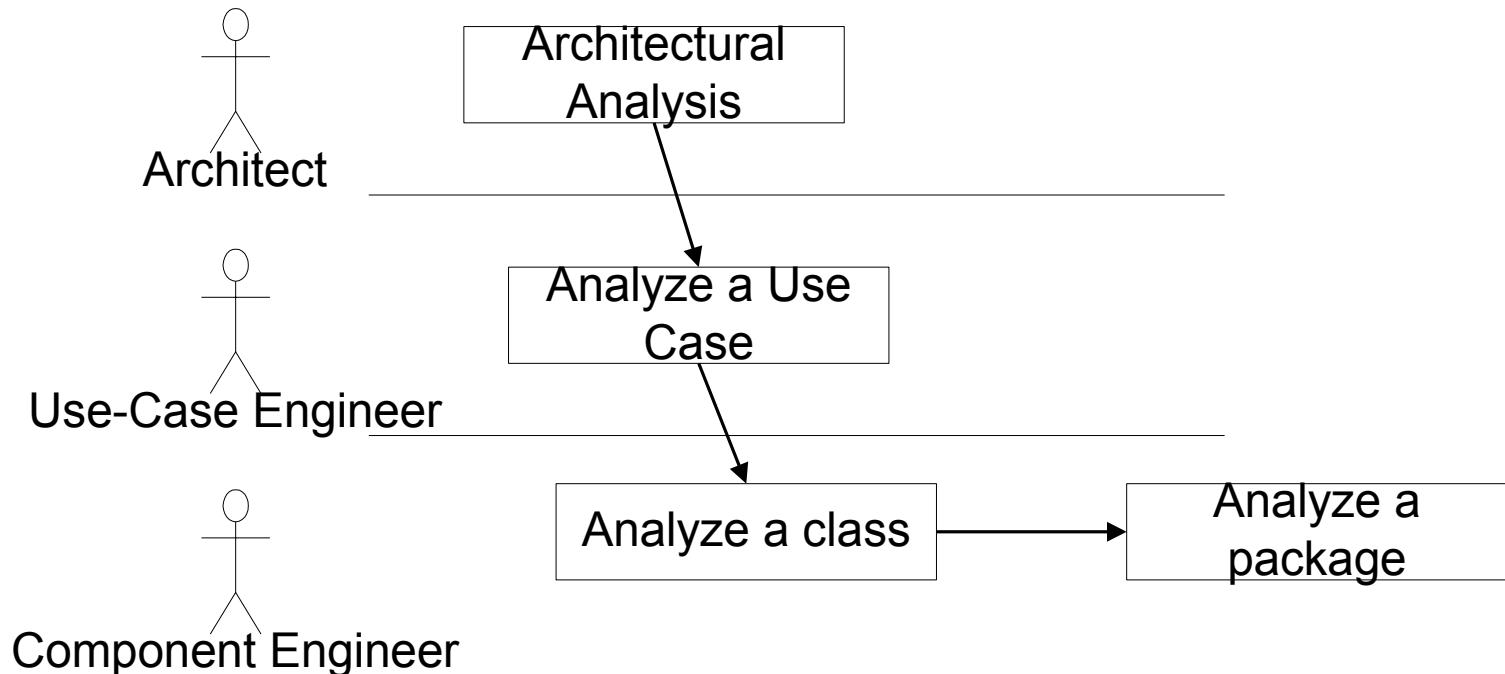


Packages and dependencies



- Reduce coupling
- Increase cohesion
- In packages
 - Cohesion is between classes in a package
 - Coupling is between classes in different packages
- In classes
 - Cohesion is between methods in a class
 - Coupling is between methods in different classes

Workflow for analysis



Architect



- Responsible for the integrity of analysis model
 - Makes sure packages fit together
 - Makes sure each package is good
 - Identifies obvious entity classes
 - Lets other classes be defined during use-case realizations and component analysis

Architect



- Identify common special requirements
 - Persistence
 - Distribution and concurrency
 - Security
 - Fault tolerance
 - Transaction management

Use case engineer



- Identify analysis classes needed by use-case
 - Boundary classes, control classes, entity classes
- Distribute behavior of use-case to classes
- Make use-case realization: a precise description of use-case
 - Sequence diagram
 - Communication diagram (in UML 2.0; collaboration diagram in UML 1.0)

Component engineer



■ Analyze classes

- Gather information from use cases
- Make sure class is coherent
- Make model as simple as possible, but no simpler

■ Analyze a package

- Relationships between classes
- Relationships between packages

Outline of RUP process for analysis



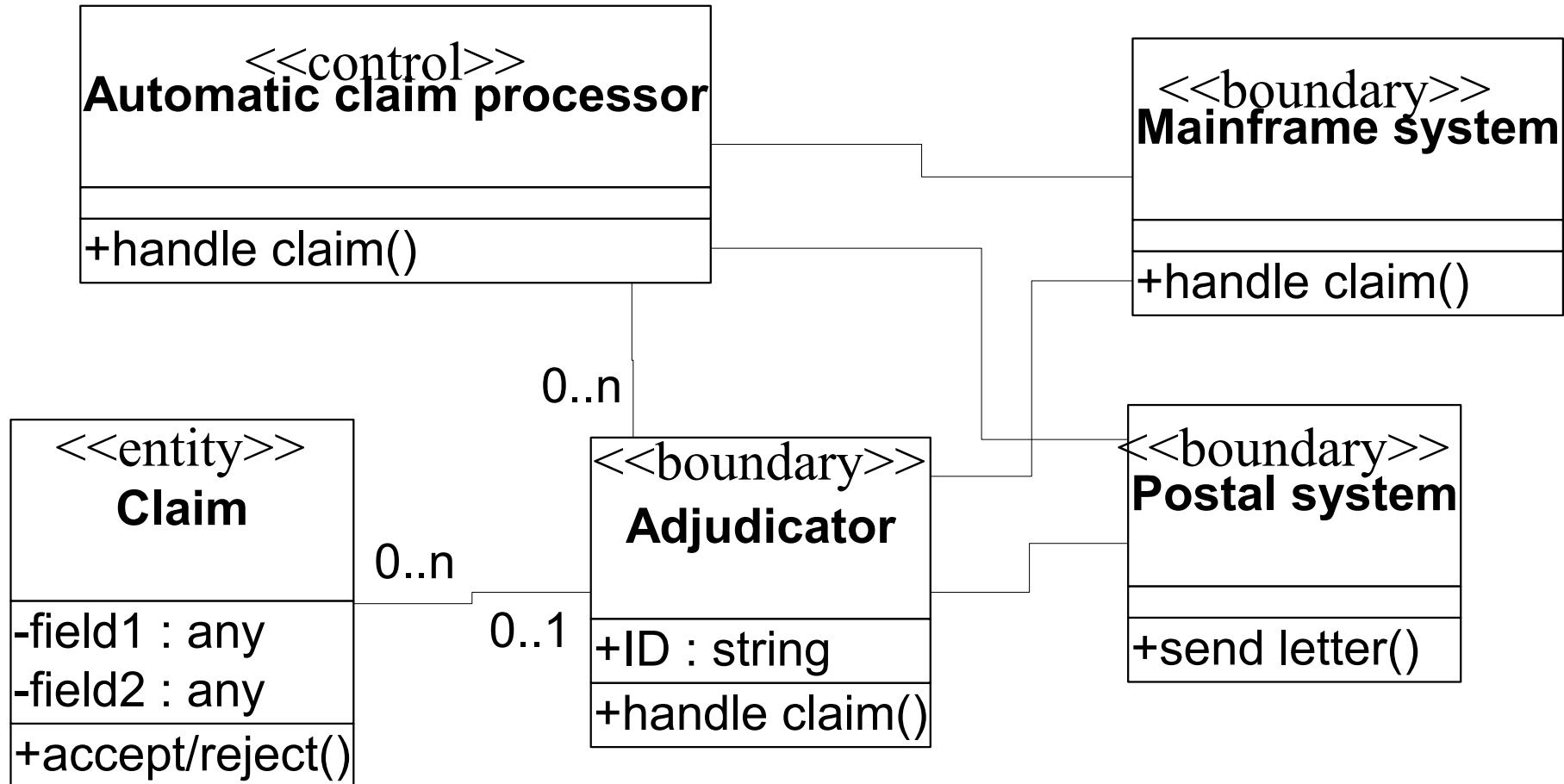
- Find use cases
- Architect determines order
- Repeatedly,
 - Take next use case
 - Change class diagram to accommodate use case
 - Simplify class diagram

Object diagram

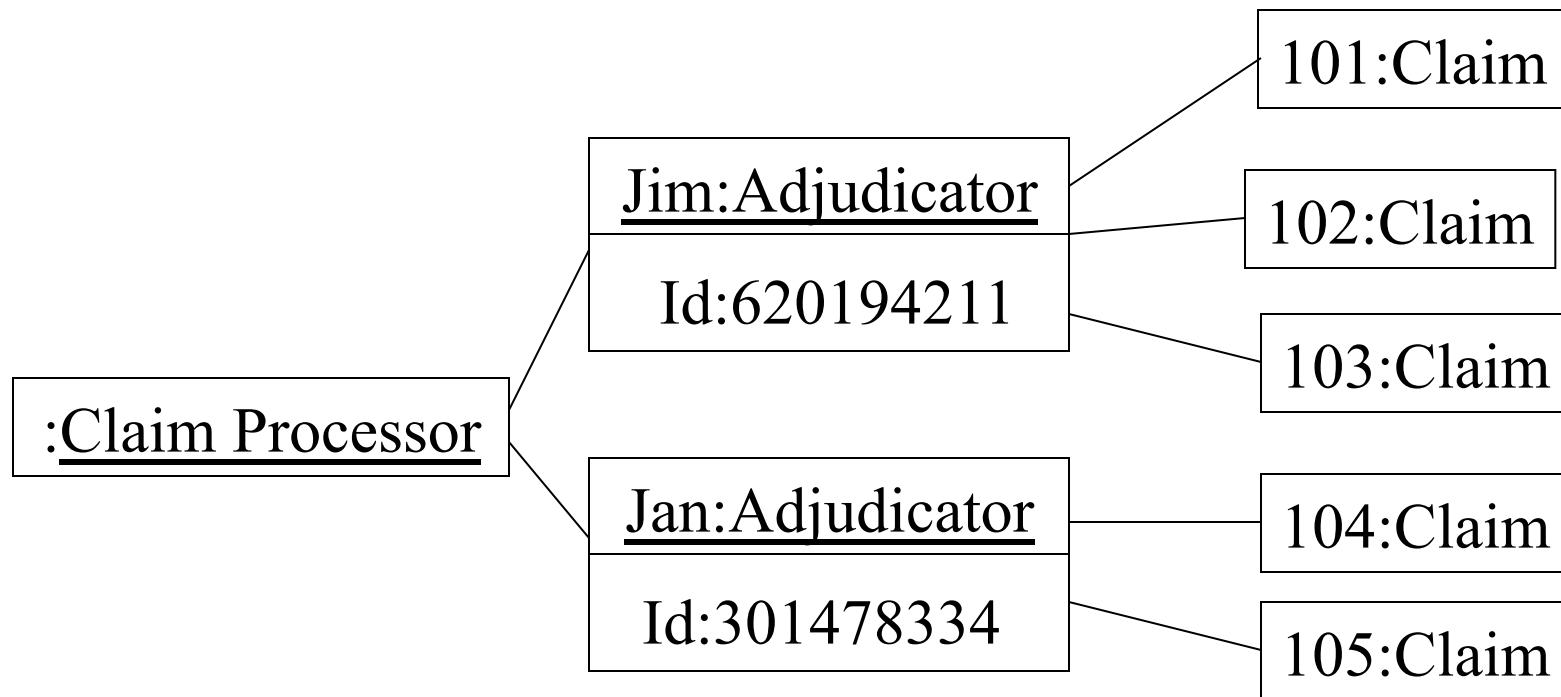


- Snapshot of objects in a system at a point in time
- If there is just one object of each class, the class diagram and the object diagram are the same
- As classes become more reusable, object diagram becomes more interesting

Example class diagram



Example object diagram



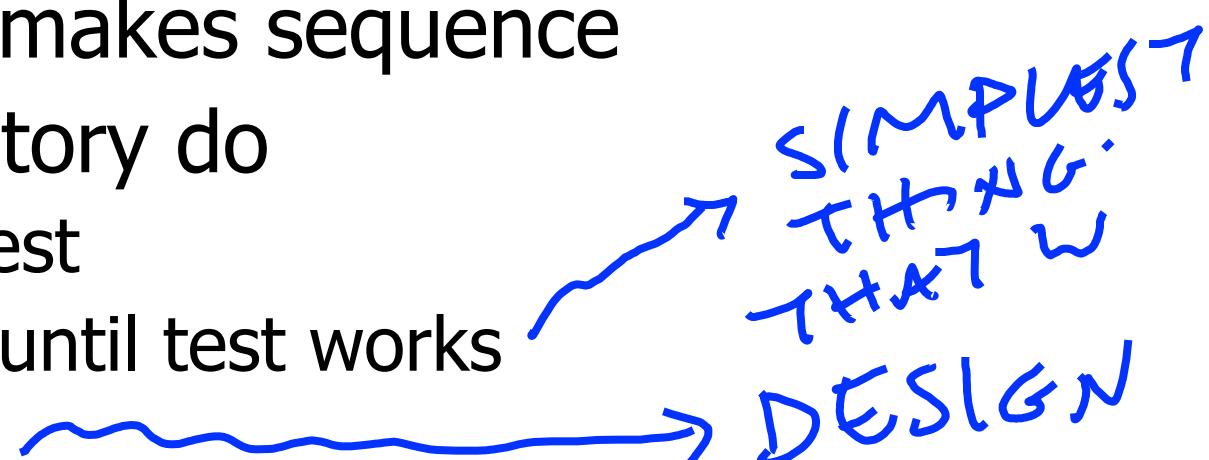
Summary



- Analysis is converting vague user needs into a precise model of what the system should do
- Analysis is incremental; look at one piece of the problem at a time
- Requires continually changing the model until you are done

XP

- Customer writes set of stories
- Developers estimate time to implement
- Customer makes sequence
- For each story do
 - Write a test
 - Program until test works
 - Refactor
 - Until there are enough tests



Next: Exam, specifications



- Oct 10 (Tue): Midterm exam
 - 7pm in 1404 SC (not DCL)
- Oct 12: Chapter 9 of Hamlet and Maybee

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Midterm exam



- Your opinion?
 - Hard/easy? Long/short? Surprising/not?
 - Reading books is important
 - Did you like format of questions? T/F?
- Grades will be on Wiki by Friday, Oct 13
 - If you have questions, please let us know
 - If you think we erred, please let us know

Project groups



- Only groups 8 and 27 have 8 students so far
 - 8 students is ideal; it may be 7 or 9 at the end
- Project fair
 - 5pm on Friday, Oct 13, room TBA (Wiki)
 - You should come if
 - You're a representative of a group with <8 students
 - You're still looking for a group
 - Each representative will give a brief presentation
 - HW2 (due Oct 24) asks for details of your project

Project grading



- You are graded on how well you follow your process (you decide XP or RUP)
 - You must know it
 - You must follow it
 - You must prove you follow it
- Make a log of what you do every day
- Project is 35% of grade
 - HWs help you to make progress on the project

Classic software engineering phases

- Put them in correct order: code, design, maintenance¹, requirements², specification, test [ME: almost all correct answers]
⁶
⁵
- We covered requirements and some design
- Today: specifications (specs)
- Do customers and users need to understand requirements (or specs)? [ME: many wrong]

T/F

T/F
CS427

Why phases?

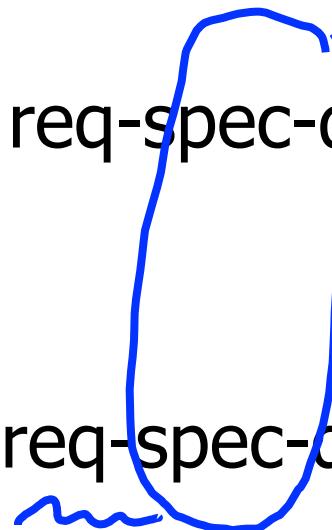


- Break big job into smaller steps
 - Let people specialize
 - Provide check-points
 - Provide early feedback
 - Provide multiple views

Other forms of decomposition



- Modules/components
 - For each module, follow req-spec-design-code-test cycle
- Features
 - For each feature, follow req-spec-design-code-test cycle



A ~~ALL~~ E ~~exists~~ Specification

- Define system as a set of data items and operations on that data
- Specification is set of properties of data and operations
- Example: bank balance is never negative

- Formal spec:

“forall a: Account. a.balance ≥ 0 ”

$\forall a \in \text{Account}: a.\text{balance} \geq 0$

Notations



- English
 - Easy to read
 - Hard to analyze
- Programming language
 - Easy for programmers to read
 - Often not powerful enough
- Special (formal) specification languages
 - We don't teach much on this topic in CS427

Why formal languages?

- Can describe many artifacts (specs, designs, requirements) [ME: many correct answers]

- Documentation (precise, unambiguous)

- Enables (automatic!) machine reasoning

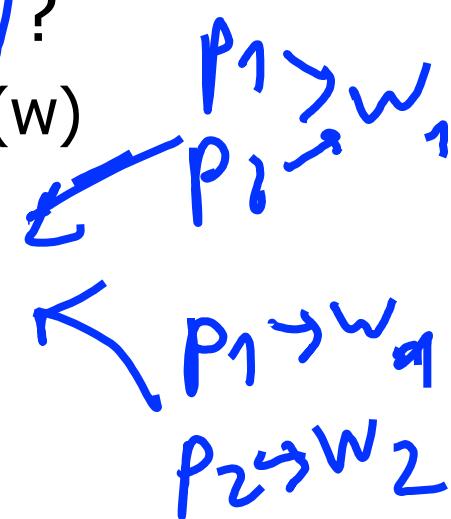
- Informal: “everybody likes a winner”?

→ ~~1) exists w: Winner. exists p: Person. p.likes(w)~~

→ ~~2) exists w: Winner. all p: Person. p.likes(w)~~

→ ~~3) all p: Person. exists w: Winner. p.likes(w)~~

→ ~~4) all p: Person. all w: Winner. p.likes(w)~~



Credit card



- A credit card has a **balance** that is the sum of all charges minus the sum of all payments. If the balance is not paid within 20 days of billing, 1% of the unpaid balance is charged as interest. Each credit card has a particular day on which it is ~~billed~~ ^{due} each month. Customers cannot charge if the charge would put balance over the limit. [Data and operations?]

Data and operations

- Credit card has

- balance
- limit
- billing day

INTEREST?
DEPENDS
ON DATE
1^o ← UNPAID
BALANCE

- Operations on credit card

- pay(amount)
- charge(amount)

2^o → DATE

Properties of operations

- Can't say balance equals sum of all charges minus sum of all payments due to interest
 - Invariant: what holds in all states
- Can say “c.pay(a).balance = c.balance - a”
- Postcondition: what operation establishes
- Can say “if c.balance + a < c.limit then c.charge(a).balance = c.balance + a”
- Precondition: what operation assumes

Question not on ME

PRE: a is sorted

- Consider binarySearch(int[] a, int v) method
- Input: array and a value to search for
- Output: position of value in the array
- Uses binary search $\leftarrow O(\log n)$
- What should be precondition?
 - a is sorted $O(n)$
 - a is not null

T/F must
T/F may

Extending operations



- “If the balance is not paid within 20 days of billing, 1% of the unpaid balance is charged as interest.”

*MBIGBUS?

First approach

- Add “unpaid balance” to a credit card
- unpaidBalance = balance on billing date
- payments get subtracted from unpaidBalance
- 20 days after billing, compute interest and charge it

→ CURRENT BALANCE

Second approach



- Add date to payments and charges
- Operations on credit card
 - pay(amount, date)
 - charge(amount, date)
- Credit card has
 - balance, limit, billing day
 - set of payments
 - set of charges

HISTORY

SPEC & IMPLEMENTATION

CODE

Second approach

"WHAT"

"HOW"

For each credit card

balance(date) =

(sum a for d <= date of charges(a, d)) -
(sum a for d <= date of payments(a, d))

There is a c in charges(a, billingDate + 20)

where ↑

c.balance(billingDate) - $\sum_{\substack{a \\ \text{charges}(a, d) \leq \text{date}}} a$
sum a for d <= date of charges(a, d)

CHARGED INTEREST

Comparison

First approach

- More like programming
- More standard notation
- Longer specs

(IMPERATIVE)

OBJECT-ORIENTED \rightsquigarrow HOW

Second approach

- Shorter, more abstract, easier to reason about

(DECLARATIVE)

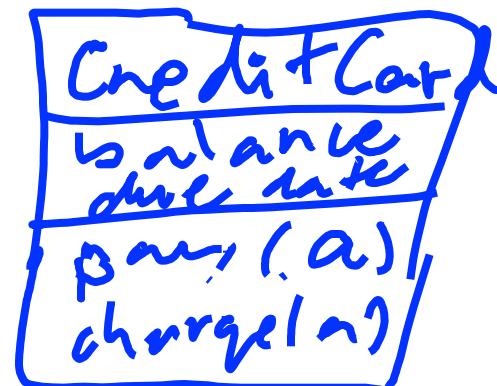
\rightsquigarrow WHAT

Personal experience: Java vs. Alloy

Design and specification



- Specification needs to know data
- Specification needs to be structured
- Must design module interfaces
- Specification is usually the design of the “entities”



Executable specifications (1)

- Some very high-level programming languages can be used to write specs
 - Prolog (Hamlet and Maybee, you don't need details)
 - Lisp
 - Smalltalk
 - Maude (several courses on formal methods)

Ch 10

QUANTIFIER \forall, \exists NON-EXEC.

Executable specifications (2)



- Write and test specification
 - Rewrite in efficient language
-
- Write and test specification
 - Rewrite to be more efficient

Story about Ada



- Ada - language invented by military project around 1980
- Specification, verification suite
- First Ada compiler written in SetL
 - “Set Language”
- Slow, but tested spec and verification suite
- Moral: executable specifications are useful

Story about concurrency



- System for downloading data from cash registers (Prof. Johnson’s experience)
- Frequent deadlock
- After fourth attempt, developed formal specification and derived a design
- Moral: formal techniques can solve hard problems
- Automated analysis: model checking

Conclusion



- Specifications can be
 - Informal or formal
 - For entire system or just a small part
 - Executable or non-executable
- You must decide what you need
 - Very common in software engineering
 - Many things are vague and ambiguous
 - Advice: use your experience (get some in CS427)

Next time



- Read chapter 10 of Hamlet and Maybee
 - Be familiar with the controversy
 - Don't need to learn details of Prolog

CS427: **Software Engineering I**



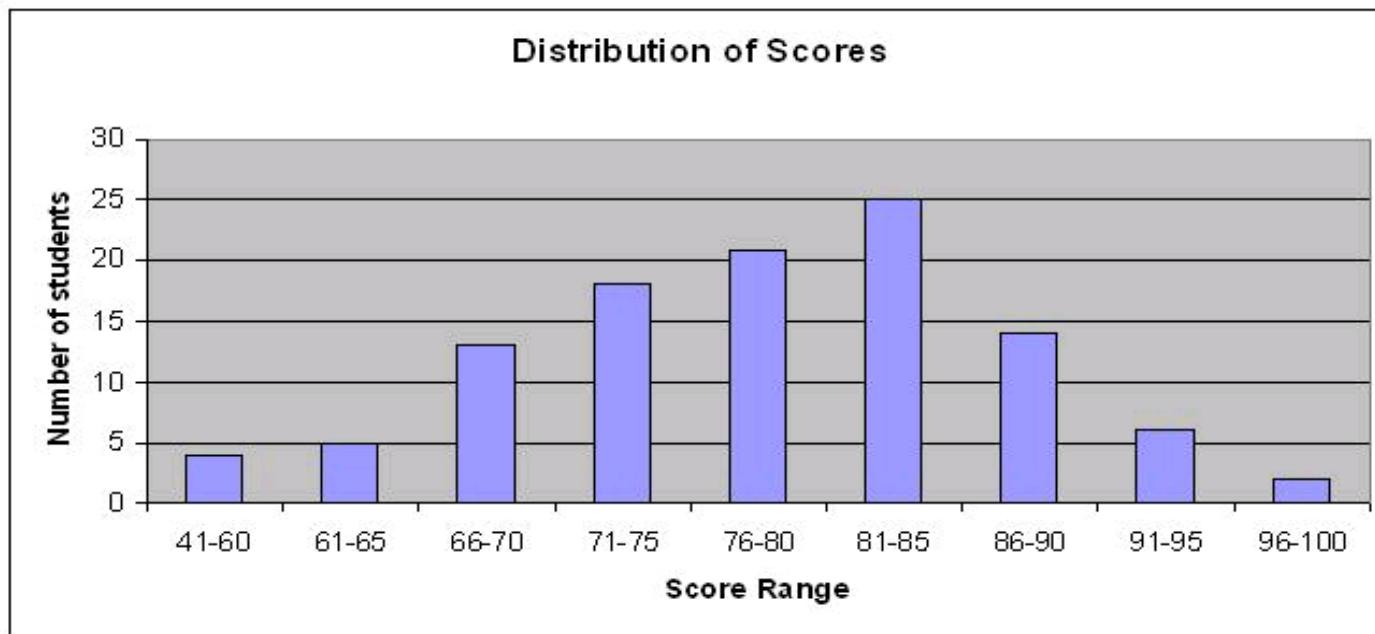
Darko Marinov

(slides from Ralph Johnson)

Midterm exam



- You can get graded exams from TAs
 - Feel free to ask if you think we made mistake
- Grades almost normally distributed



Comment on midterm exam



- | COCOMO question is picking details from the reading; is it really like that?
- | One correct answer, using formula $E \sim \text{size}^P$
 - | This is not exponential, not $E \sim P^{\text{size}}$
- | Another correct answer, common sense
 - | $E(200\text{KLOC}) = 2*E(100\text{KLOC}) + E(\text{integration})$
- | Common sense can lead to incorrect answers
 - | $E(200\text{KLOC}) \leq 2*E(100\text{KLOC})$

Project info



- Groups formed (any volunteers for 10?)
- Grading: You are graded on how well you follow your process (you decide XP or RUP)
 - You must know it
 - You must follow it
 - You must prove you follow it
 - Make a log of what you do every day
- HW2 is due next Tuesday, Oct 24
 - Use cases and UML diagrams for your project

Topics



- Covered
 - Requirements
 - Some specification
 - Some design
- Today: specification and design
 - Chapters 9, 10, 11 of Hamlet and Maybee
 - Formal specs
 - Tests as specs
 - Specs vs. design

Use formal specifications (1)



- To develop models of parts of system
 - Finite state machines, abstract data types
- To develop model of one aspect of system
 - Data-flow diagram, security, architecture
- To learn how to think about problem
 - Pre/post conditions, invariants
 - Set, sequence, mappings (generic ADTs)

Use formal specifications (2)



- For particular kinds of applications
 - Compilers
 - Grammar
 - Denotational semantics
 - Safety critical

What you should know about formal methods



- Chapter 10 of Hamlet and Maybee
- Be familiar with the controversy
- Don't need to learn Prolog

Tests as specifications



- Formal specification (entire space)
 - Balance after you add money to an account will be the balance beforehand plus the amount of money you added
- Test (one point in the space)
 - Create account with balance of \$100
 - Add \$20 to account
 - Account has balance of \$120

Advantages of tests as specs



- Concrete, easy to understand
- Don't need new language
- Easy to see if program meets the specs
- Making tests forces you to talk to customer and learn the problem
- Making tests forces you to think about design of system (classes, methods, etc.)

Disadvantages of tests as specs



- Hard to test that something can't happen
 - Can't withdraw more money than you have in the system
 - Can't break into the system
 - Can't cause a very long transaction that hangs the system
- Tends to be verbose
 - Sampling many points from the space

JUnit



- www.junit.org
- A test is a method whose name starts with “test” in a subclass of TestCase
 - Version 4.0 also allows @Tets annotations
- Calls methods (defined in TestCase) like
 - `assertEquals(object1, object2)`
 - `assertTrue(boolean)`

Testing Student and Course



```
public void testStudentCreation() {  
    Student s = new Student("Marvin Minsky");  
    assertTrue(s.getName().equals("Marvin Minsky"));  
}
```

or

```
public void testStudentCreation() {  
    Student s = new Student("Marvin Minsky");  
    assertEquals(s.getName(), "Marvin Minsky");  
}
```

Normal test



- Set up “test fixture”
- Call method that is being tested
- Assert that result is correct

More involved example



```
public void testAddingStudent() {  
    Course c = new Course("Tourism 101");  
    Student s = new Student("Donald Knuth");  
    c.addStudent(s);  
    assertEquals(c.students().size(), 1);  
    assertEquals(c.students().elementAt(0), s);  
}
```

Tests as specifications



- Tests show how to use the system
- Tests need to be readable
 - Need comments that describe their purpose
 - Keep short, delete duplication

Design



Verb

- To conceive or plan out in the mind
- To make a drawing, pattern, or sketch of

Life-cycle



- Requirements capture
- Specification
- Design
- Implementation

A good design



- Satisfies requirements
- Easy to implement
- Easy to understand
- Easy to change
- Easy to check for correctness
- Is beautiful (welcome [back] to Software Engineering)

Keep It Simple (S)



KISS

- Einstein: “Everything should be as simple as possible, but no simpler.”
- Saint-Exupery: “A design is perfect not when there is nothing to add, but when there is nothing to take away.”
- Gates: “Measuring a program by the number of lines of code is like measuring an airplane by how much it weighs.”

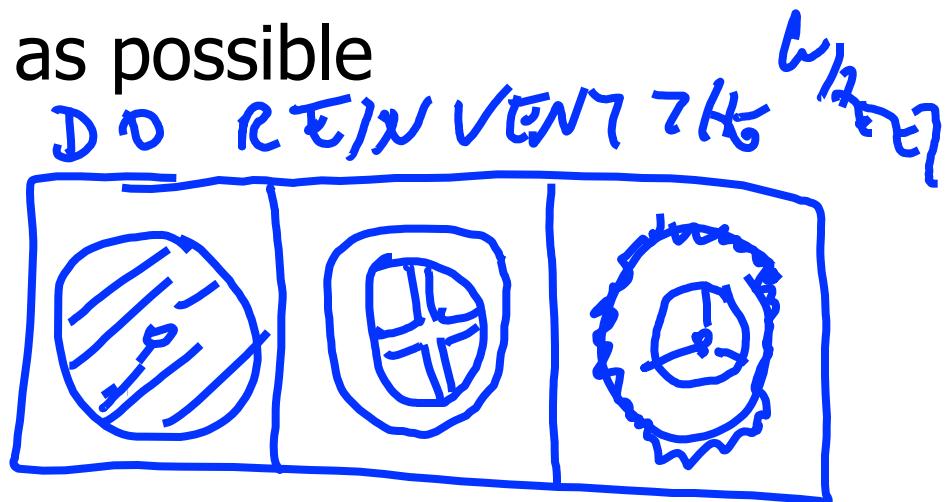
Keep it simple



- Avoid duplication
- Eliminate unnecessary features/code
- Hide information
- No surprises (follow standards)

Davis Design Principles (1)

- Don't reinvent the wheel
 - Search the web. Read the book. Talk to experts.
- Exhibit uniformity and integration
 - Make parts as similar as possible
 - Coding standard
 - Standard names

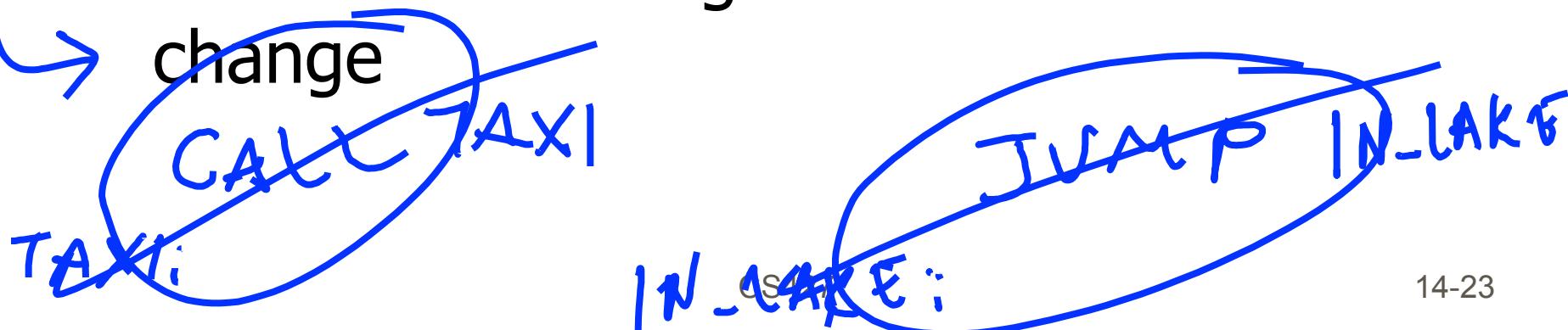


`f(multi) {
}`

Davis Design Principles (2)



- Minimize the intellectual distance between the software and the problem as it exists in the real world
 - Use same names as your customers
 - Use same models as your customers
- Structure the design to accommodate change



Davis

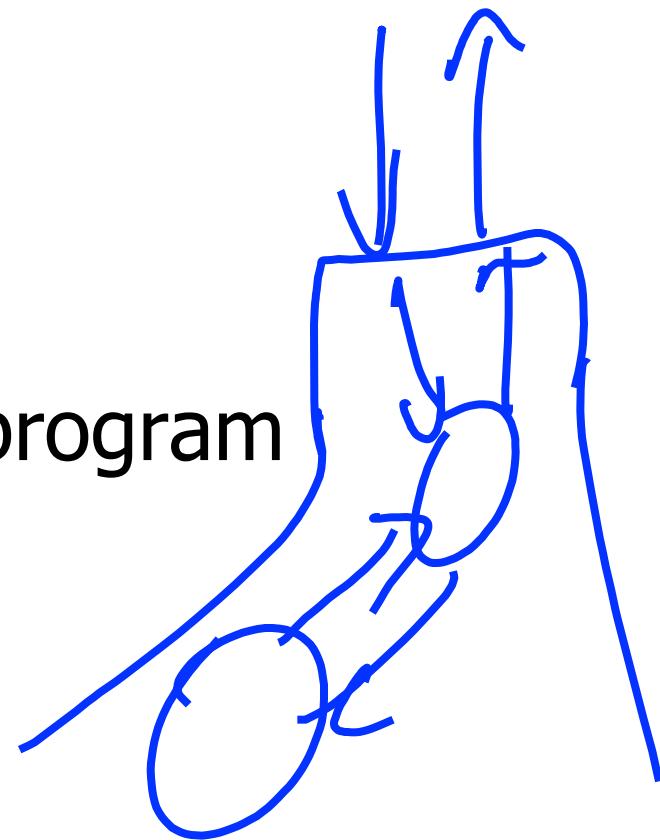


- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual errors

Many things to design



- Company
- System that uses program
- Interface to program
- Interface of module within program
- Procedure
- Database



Design



- Process of eliminating “misfits”
- Find something wrong and fix it
- Good design requires being able to spot something that is wrong

What can go wrong?

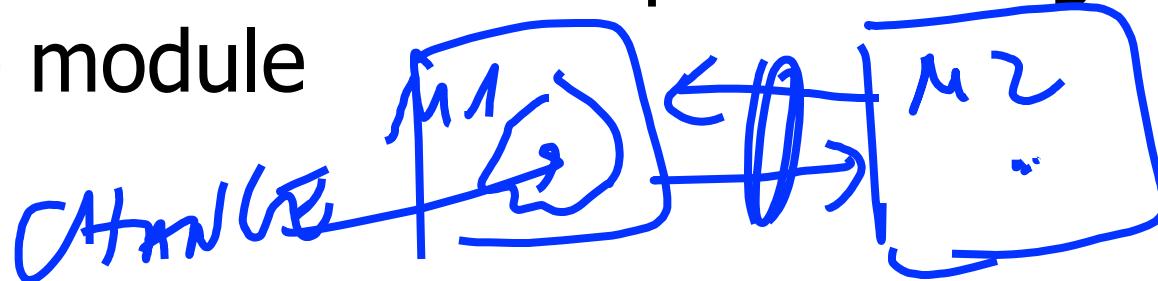


- Too complex - can't understand
 - Users can't understand
 - Programmers can't understand
- Too slow
- Wrong features
- Too hard to add new features
- Not compatible

Too hard to add new features



- Usually because problem is decomposed improperly
- Design decisions should be hidden
- Only one module should know about design decision
- Changing that decision requires changing only one module



Separate user interface



- Separate user interface from program logic
- UI changes frequently
- It is easier to change UI if it is separate
- UI experts can be non-programmers
- Can provide several UIs for one system

Automated tests



- UI hard to test
- Separate UI from program logic and write automated tests for program logic
- UI first?
 - No, design program logic first and write tests for it
- Database first?
 - No, design program logic first and write tests for it. Then design database and rewrite to use it

Summary



- Fuzzy boundaries between
 - Analysis and design
 - Design and implementation
- Design is problem solving
- It helps to know a lot of solutions

Project advice repeated



- You are graded on how well you follow your process.
 - You must know it.
 - You must follow it.
 - You must prove you follow it.
- Make a log of what you do every day.

Next time



- Read chapters 13 and 14 of Hamlet and
Maybee

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Grading info



- Project: You are graded on how well you follow your process (you decide XP or RUP)
 - You must know it
 - You must follow it
 - You must prove you follow it
 - Make a log of what you do every day
 - Interact with your TA
 - Don't forget the product!
- Postpone HW2? For two days? For a week?

Today: Component-level design



Requirements gathering

Analysis/Specification

Architectural design

Component-level design

Coding

Testing

“Design” : Noun



- Refinement
 - Requirements
 - High-level design
 - Details
- Information-hiding
 - Don’t show everything
 - Reveal bit by bit
- Modularity

“Design” : Verb



- Bounce from high-level to low-level
- New requirements come after design is created
- Design is created incrementally
 - As requirements are known
 - As better design ideas are invented
 - As design flaws are discovered

Component design



Numerous techniques

- Flow charts
- ✓ Decision tables
- ⊕ Pseudo-code (Ch. 13 of Hamlet and Maybee)
- ⊕ State machines (Ch. 14 of Hamlet and Maybee)

Decision tables



- Used to specify program with complex conditions
- Make it easy to see if any cases are missing
- Can be implemented with IF statements

Example requirement



IRS Publication 946: “The recovery period is 27.5 years for residential real property, 31.5 years for nonresidential real property placed in service before May 13, 1993, 39 years for non-residential real property placed in service after May 12, 1993, 50 years for railroad gradings and tunnel bores, except that nonresidential real property on an Indian reservation has a recovery period of 22 years.”

Decision table for recovery period

RECOVERY

Conditions	1	2	3	4	5
Real property	T	T	T	T	T
Residential	T	F	F	F	F
Placed before May 13, 1993	?	T	F		
Railroad grading or bore	?	F	F	T	?
On Indian reservation	?	F	F	F	T
Actions	OUTCOMES				
27.5 years	X				
31.5 years		X			
39 years			X		
50 years				X	
22 years					X

Pseudocode



- Also known as “Program Design Language”
- Advantages
 - Expressive and compact
 - Can use any editor
 - Sometimes can type-check/compile it
- Disadvantages
 - Must know the language

~~Java~~ Pseudocode

```
recoverPeriod(property) {  
    if (isReal(property)) {  
        if (isResidential(property)) return 27.5;  
        if (onReservation(property)) return 22;  
        if (isRailroad(property)) return 50;  
        if (property.date > "May 13, 1993" )  
            return 31.5;  
        else return 39;  
    }  
    ...  
}
```

Pseudocode

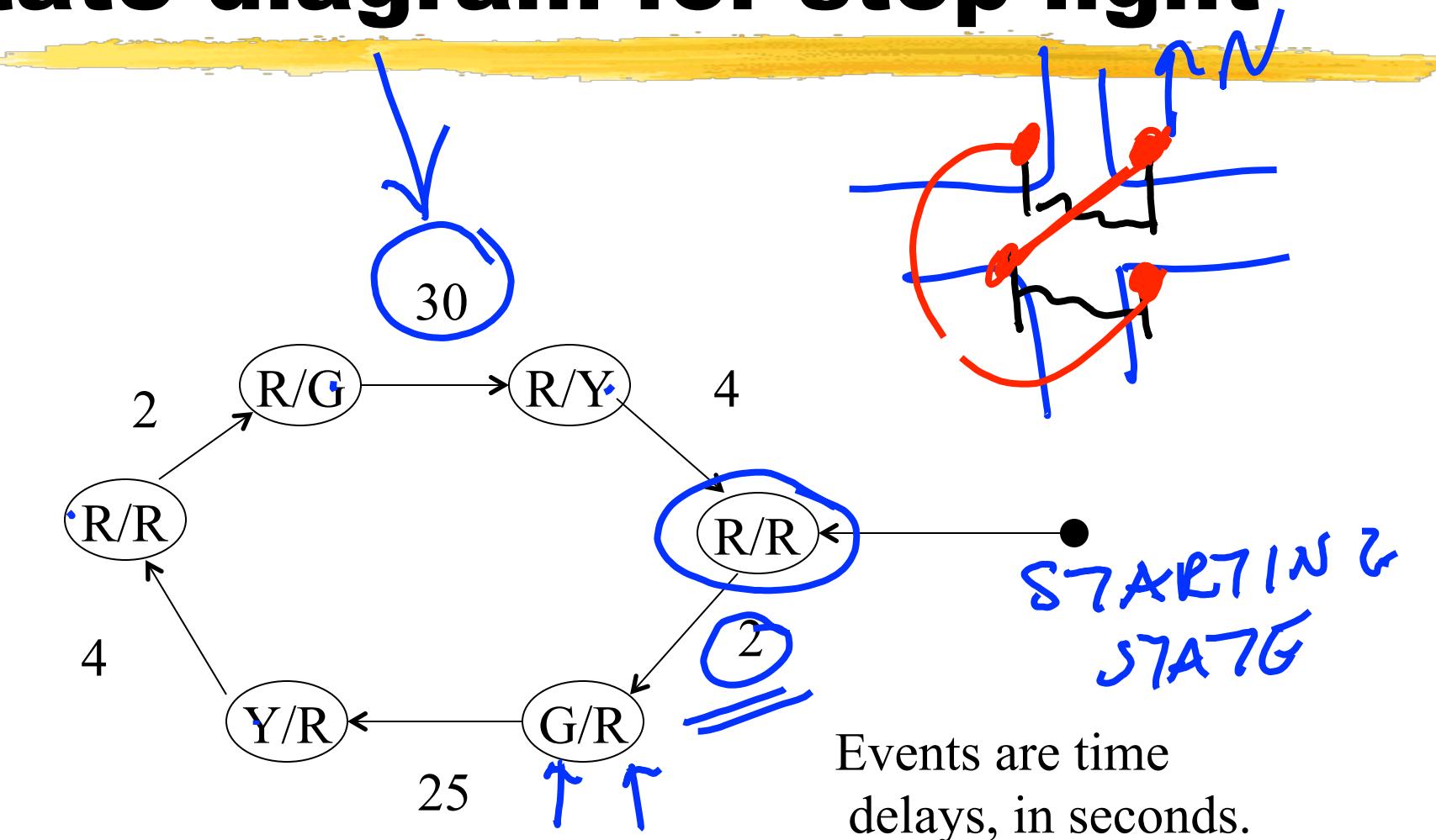


- Works well with refinement
- Write pseudo-code
 - As comments
 - With stubs
- Gradually implement it all
- Execute and test as you go

State machines (FSM)

- Lots of theory
 - How to minimize number of states
 - How to merge state machines
 - How to tell whether two state machines are equal
- Can generate code directly from state machines
 - But usually do not

State diagram for stop light



Pseudocode for stop light

Action = Record {integer wait, east, north}

Action: Actions[1 .. 6]

repeat forever

 for I = 1 to 6 do

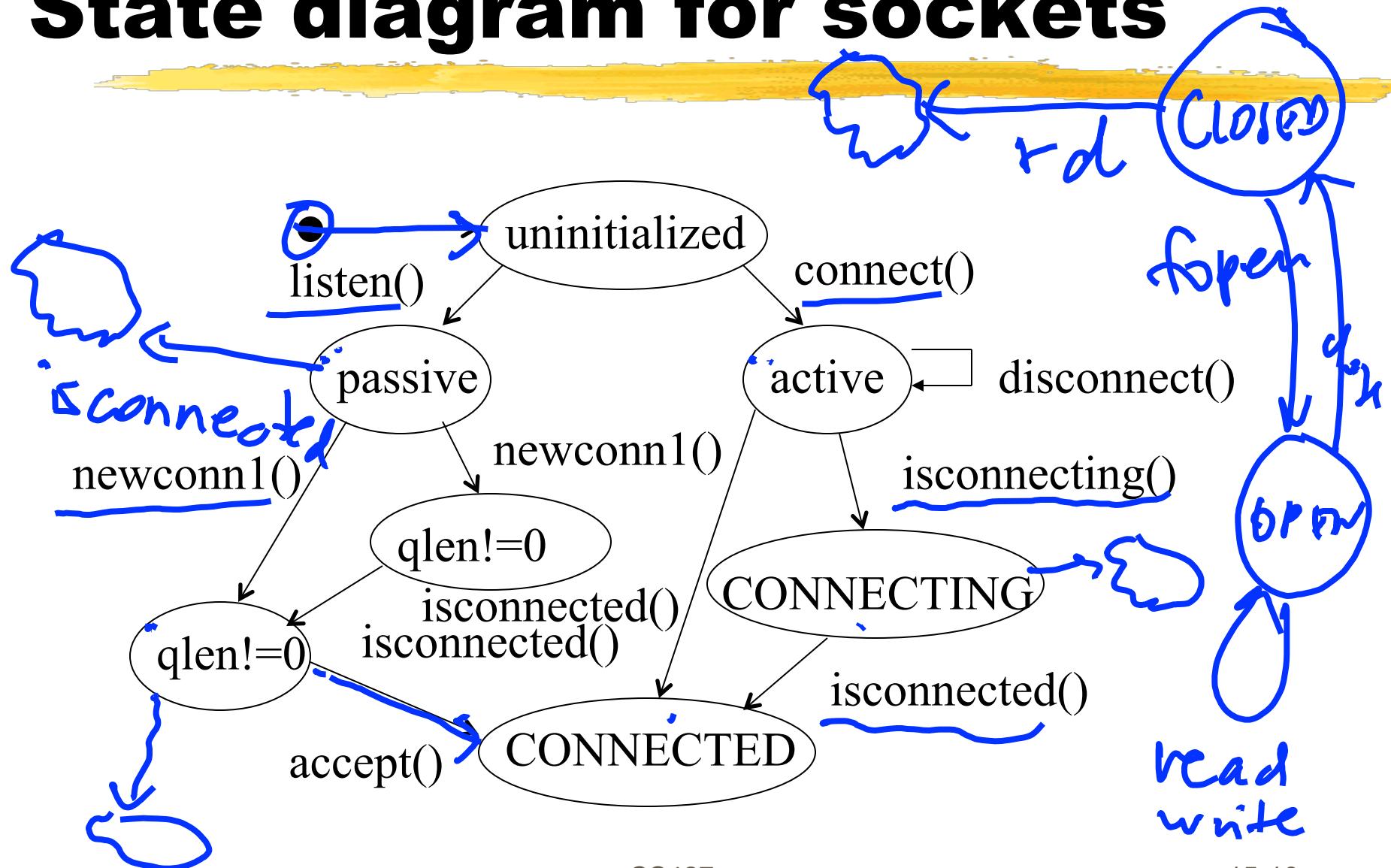
 setEast(actions[i].east)

 setNorth(actions[i].north)

 wait(actions[i].wait)

 end for

State diagram for sockets



Implementing socket

- Socket is an object
- Actions are methods of the socket
- State is stored in variable of the object
- Each method uses IF statement to make sure socket is in the right state
- When IF statements get too complicated, use “State Pattern”

State pattern

ACTION | STATE

From “Design Patterns”

Socket

SocketState
listen()
connect()
newconn1()
...

ConnectingState

ConnectedState

...

PassiveState

IS Connected () {
if (state == PASSIVE)
if throw new Exception...
if if if

Detailed design



- Lots of different techniques
- Most only works in a few places
- Pseudocode works in most places, but often there is a better technique
- Often best to use code and skip detailed design

Design in XP



- No required design documents
- Developers talk about design, but write test cases and code
- Need to design during:
 - Estimating user stories
 - Iteration planning (making engineering tasks)
 - At start of programming task
 - When task does not go well

Design in XP



Much of the design created during refactoring

- | Simple design
 - | Code “smells”
 - | Coding standards
-
- | Code communicates design



Keep system simple



■ Small classes

- 7 methods is very nice
- 20 methods is a little large
- 80 methods is horrible

■ Small methods



- Smalltalk: 20 lines is large
- Java: 40 lines is large
- C++: 60 lines is large

Keep system simple



- Decompose large classes
 - Variables that change together should stay together
 - Methods should be in class of variables that they access
 - “Ask, don’t tell.”

Keep system simple



- Decompose large methods
 - Turn repeating code into new methods
 - Turn loop bodies into new methods
 - Turn complex conditions into new methods
 - Turn logical blocks into new methods, hiding temporaries

Code (design) smells



- Large classes, methods, packages
- Duplication
- Classes with no methods except accessors
- One class with all the code
- Complex conditionals and no polymorphism

Good design



- Good design is product of many small steps
- “Do the right thing” — VALIDATION
 - Each step adds a feature
 - Do one more thing right
- “Do the thing right” — VERIFICATION
 - Each step makes design a little simpler
 - Eliminates one more unnecessary complexity

Next: Modularity and abstraction



Read:

<http://www.acm.org/classics/may96/>

D.L. Parnas “On the Criteria To Be Used
in Decomposing Systems into Modules”

Optional:

<http://www.tao.com/pub/abstraction.txt>

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Comments on class



- THANK you to everyone who submitted
- Positive: S.E. talks about real practice
 - “I like how I see a lot of the processes I experienced on past internships--but never knew they had titles”
 - “The real world examples [given] in class are useful.”
 - “I can use [the processes] at my real job, and show how process and techniques have been made to fit within the real world.”

More comments



- Controversial: amount of reading
 - “Too many books! Pick one or two AT MOST.”
 - “We need more reading.”
- Controversial: usefulness of reading
 - “The midterm did not test the material from the 500+ pages of reading, or the lecture notes. People who attended class and read the books did the same as people who never attended and never read.”
 - “The class seems to rely more on the books rather than the actual lectures. If you read the books you're fine, otherwise you're not.”

S.E. is broad but shallow



- Many processes to develop SW
 - E.g., video games vs. banking application vs. flight-control software
 - Everything we teach (through lectures or reading) will be useful to some of you
 - None of you will need everything we teach
- Example at interviews: UML, other topics?
 - How many lectures to cover all details of UML?
 - Can we cover everything in lectures?

Communication



- If you have questions, please do feel free to ask!
- You can attend office hours
 - Jeff on Mondays 3-4
 - Valas on Tuesdays 10-11
 - Darko on Thursdays 2-3
- You can post on the newsgroup
- You can email any of us
- Do you want anonymous communication?

Homework 2



- Postponed for a week: due Tue, Oct 31
- Use this time to work more on the project
 - Project is more than homework assignment
 - Project asks you to follow a process
 - Homework asks you to do some of the steps

Topics on design



- Previous: component-level design
- Today: modularity and abstraction
 - <http://www.acm.org/classics/may96/>
 - Optional: <http://www.tao.com/pub/abstraction.txt>
- Next: refinement
 - Hamlet and Maybee, chapter 15
 - Two optional papers
- Optional: exam won't ask you anything beyond lectures (but interviews?)

Modularity



Myers 1978: “Modularity is the single attribute of software that allows a program to be intellectually manageable.”

Split a large program into smaller modules

What is a module?



- Procedure
- Class
- File
- Directory

Functional independence



- Cohesion: Each module should do one thing
 - We want high cohesion
- Coupling: Each module should have a simple interface
 - We want low coupling

Cohesion



- Measure of interconnection within a module
- The degree to which one part of a module depends on another
- Maximize cohesion

Types of cohesion



- Coincidental – avoid meaningless relations
- Logical - same idea
- Temporal - same time

- Procedural - calls
- Communicational - shared data

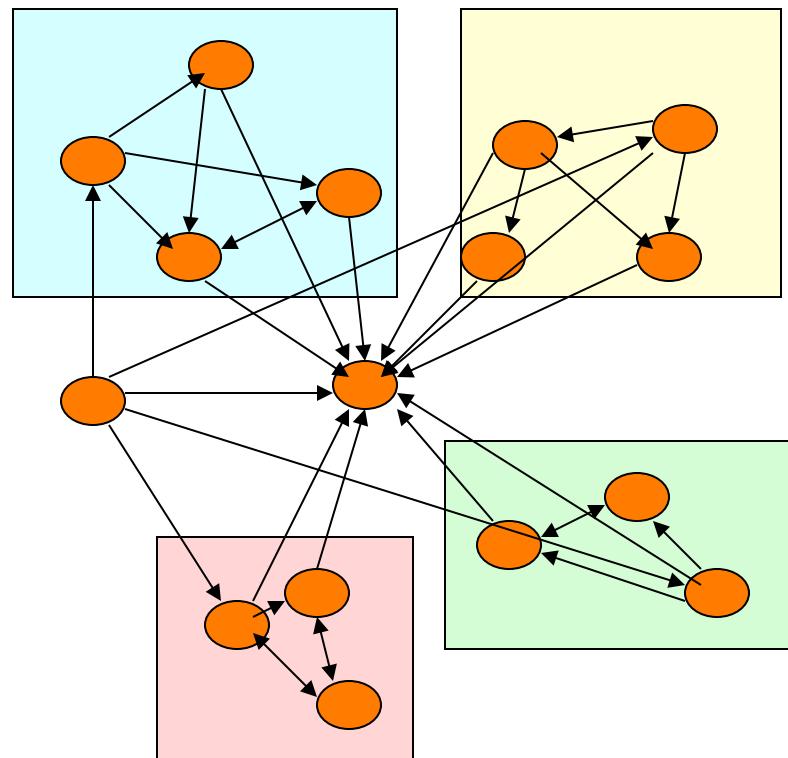
- Change together

Coupling

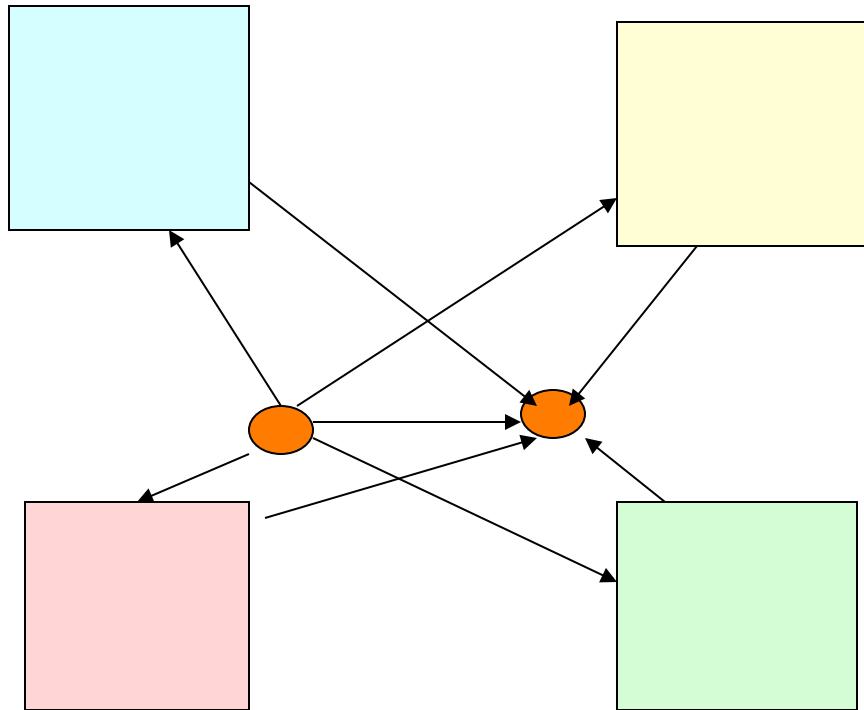


- Measure of interconnection among modules
- The degree to which one module depends on others
- Minimize coupling

Modularity



Interfaces



Information hiding



- Each module should hide a design decision from others
- Ideally, one design decision per module, but usually design decisions are closely related
- D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems Into Modules," CACM, Vol. 5, No. 12, December 1972

Trade-offs



Suppose moving functionality from one module to another will

- Decrease coupling between modules
- Increase cohesion in one module
- Decrease cohesion in another

Should you do it?

How to get modularity



- Reuse a design with good modularity
- Think about design decisions – hide them
- Reduce coupling and increase cohesion
 - Refactor: make small changes
- Reduce impact of changes
 - If adding a feature requires changing large part of the system, refactor so change is easy

Modularity: summary



- Modularity is key to managing complexity
- Modularity is about managing dependencies
- Modularity should be based on hiding design decisions
- No perfect answer
 - Trade-offs
 - Iteration

Abstraction



Abstract

- 1: disassociated from any specific instance
- 2: difficult to understand

Abstraction: ignoring unimportant details and focusing on key features, usually repeating

Example of abstraction



- Unix file descriptor
 - Can read, write, and (maybe) seek
 - File, IO device, pipe
-
- Which network abstraction did we cover in the previous lecture?

Kinds of abstraction in S.E.



- Procedural abstraction
 - Naming a sequence of instructions
 - Parameterizing a procedure
- Data abstraction
 - Naming a collection of data
 - Data type defined by a set of procedures
- Control abstraction
- Performance abstraction

Abstract data types (ADTs)



Complex numbers: +, -, *, real, imaginary

Queue: add, remove, size

Degrees of abstraction



- C: Queue module specifies contents
- C++: Template classes let you instantiate many Queues, each with a different type of content
- Smalltalk: Queue class can contain any kind of object

Interfaces



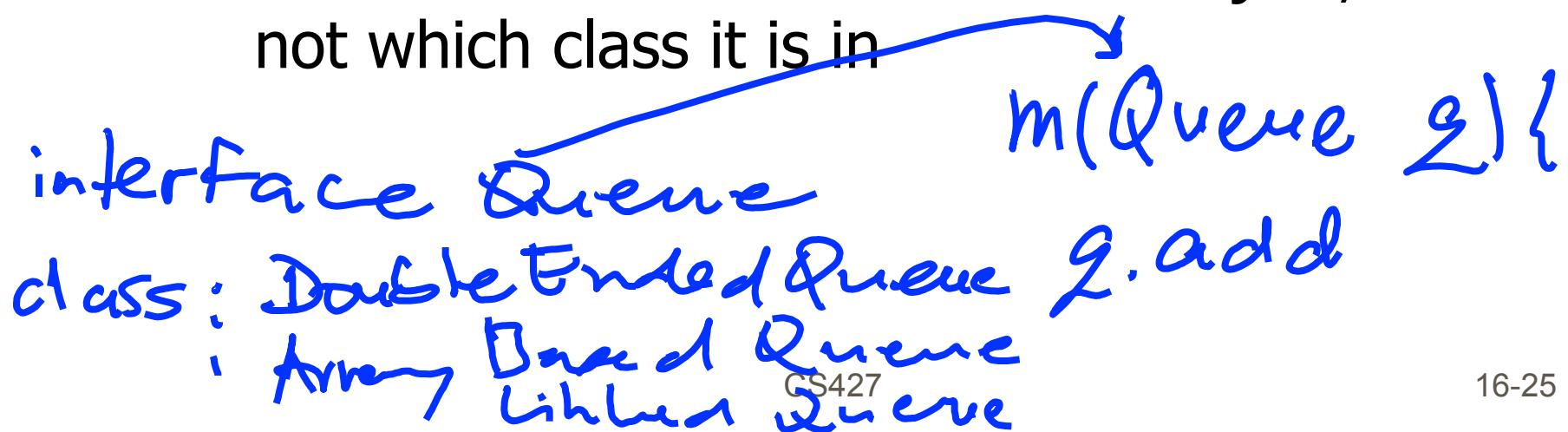
- Modules but not objects
 - Client knows which module it is using but not details of its implementation
- Object-oriented programming
 - Client knows the interface of an object, but not which class it is in

interface Queue

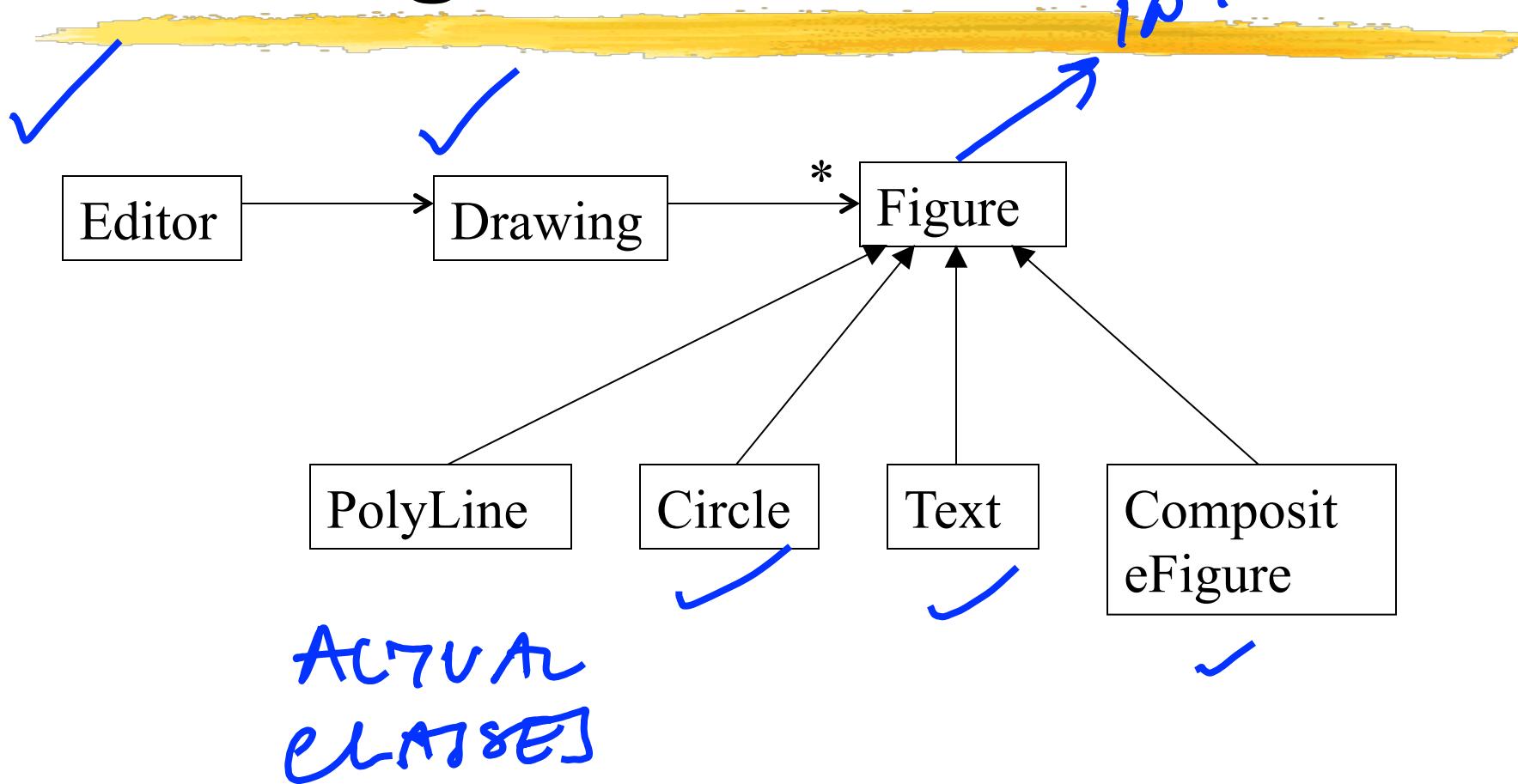
class: DoubleEndedQueue q.add

: Array Based Queue

Lihua CS427 Queue



A drawing editor



Facts about abstraction



- People think about concrete details better than abstractions
- People learn abstractions from examples
- It takes many examples to invent a good abstraction
- Abstractions are powerful, but focus on facts first

How to get good abstractions

- Get them from someone else
 - Read lots of books
 - Oops, is it okay to suggest reading?
 - Look at lots of code
- Generalize from examples
 - Try them out and gradually improve them
- Eliminate duplication in your program

Abstractions can fool you

Suppose collection has operation

getItemNumbered()

How do you iterate?

For $i := 1$ to length , getItemNumbered(i)

But what if collection is a linked list?

How much would the iteration take?

$l.\text{iterator}().\text{next}()$

$$\sum_{i=1}^{\text{len}} i = O(\text{len}^2)$$

Abstraction: summary



- Modules should implement abstractions
- Best modularization is based on ADTs
- Not all abstractions can be implemented by modules

Next: Refinement

ADTs

- Hamlet and Maybee, chapter 15
- Two optional papers
 - To repeat what “optional” means: exam won’t ask anything beyond lectures (but interviews?)

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- Wiki is down again
 - My apologies for that!
 - Jeff will move Wiki as soon as he's back
- Course web page has lecture slides and links to videos
 - <http://www.cs.uiuc.edu/class/fa06/cs427/>
- We'll get everything on one server in the future (hopefully a reliable server)

Topics on design



- Previous
 - Component-level design
 - Modularity and abstraction
- Today: Refinement
 - Hamlet and Maybee, chapter 15
 - Two optional papers
- Next: Object-oriented design or more XP?
 - Hamlet and Maybee, chapter 16
 - or no reading?

Some previous topics



- Decision tables
- Pseudo-code
- State machines

Previously: Modularity



- What is it?
- What are the two key concepts?

Some previous topics



- Information hiding

- Abstraction

Today: Refinement



- Alan Perlis: Epigrams on Programming
 - <http://www.cs.yale.edu/quotes.html>
- 15. Everything should be built top-down, except the first time.

Abstract Data Type (ADT)



- Formalism based on solid mathematics
- Define a type as
 - A set
 - A set of operations on that set
 - A set of properties that the operations must have

Complex Numbers



Complex: Real X Real

=: Complex X Complex -> Boolean

+: Complex X Complex -> Complex

-: Complex X Complex -> Complex

*: Complex X Complex -> Complex

real: Complex -> Real

imag: Complex -> Real

Axioms



$\forall c, d: \text{Complex}$

$$\text{real}(c+d) = \text{real}(c) + \text{real}(d)$$

$$\text{imag}(c+d) = \text{imag}(c) + \text{imag}(d)$$

$$\text{real}(c*d) = \text{real}(c) * \text{real}(d) - \text{imag}(c) * \text{imag}(d)$$

$$\begin{aligned} \text{imag}(c*d) = & \text{real}(c) * \text{imag}(d) + \\ & \text{imag}(c) * \text{real}(d) \end{aligned}$$

$$(\text{real}(c) = \text{real}(d) \wedge \text{imag}(c) = \text{imag}(d)) \Leftrightarrow c = d$$

Abstract Data Types



- Based on math - no side effects
- Easy to prove things about an ADT
- Programming languages (Modula, Ada) had variables and side effects
- Object-oriented programming based on objects with identity and with changeable state

ADT vs. Objects



- Value Objects [advanced question for ME]
 - Object that is really an ADT
 - Instance variables set ONLY by constructor
 - Return a new object, don't change the old one
 - Value Object never has problems with side effects, can be freely copied, especially useful for database or distributed system

ADT vs. Objects



- ADT is the value of an object
- Example: Queue
- Operations: add, remove, length, first

- Specify a Queue with an ADT
“Sequence”

Specify Queue



{state = s}

Queue::Add(Element e)

{state = add(s,e)}

$\forall s: \text{Sequence: } \text{length}(\text{add}(s,e)) = \text{length}(s) + 1$

Refinement



Design developed top-down by repeatedly adding detail to a high-level design

- Original based on procedures
- Also based on ADTs or objects
- Also based on formal specifications
- Also based on use cases->class diagram->event diagram->code

Refinement



- Designing is a sequence of steps
- Each step transforms the design
- Each step is a design decision
- Documenting each step lets reader check the design

Example: procedural refinement



- One optional paper at
<http://www.acm.org/classics/dec95>
mostly procedural refinement
- Step consists of defining an undefined procedure call, possibly by calling some new undefined procedures
- Start with single high-level procedure
- Finish when all procedures are defined

Example: ADTs or Objects



- Define entire system with a few ADTs (classes)
- At first, only define the set of values and operations (instance variables and method interface)
- Specify operations, possibly by introducing new ADTs (classes)

Example: RUP



- Write use case diagram
- Write use cases
- Develop class diagram, interaction diagrams
- Develop packages
- Write code for classes in each package
- Test each class, each package, entire system

Refinement myths



You make one pass through the design

Each step adds something to the design

Each step is independent of the others

You iterate because you don't have a methodology

Refinement facts



- Designers bounce between high-level and low-level concerns
- Designers sometimes redo previous steps
- Steps often done “out of order”

Rational design



“A Rational Design Process: How and Why to Fake It” by David Parnas and P.C. Clemens, *IEEE Transactions on Software Engineering, SE-12:2, February 1986.*

Rational design process - one in which every step has a reason

Top-Down design is a myth



- Requirements change
- Requirements are incomplete
- Existence of system changes requirements
- Analysts make mistakes
- Designers make mistakes
- Time reveals better solutions
- Too many decisions to note them all

How and Why to Fake It



- Documentation should be structured so that it describes a sequence of refinements with a reason for each
- Sequence not required to be historical
- Designers can reorganize and rewrite sequence if necessary
- Result: clear design that can be reviewed

Lessons



- Iteration is important part of real process
- Get running code as soon as possible!
- Hide effects of iteration on the documentation

Lessons



- It is easier to go from design to analysis if the same person does both
- Divide work by subsystem, not by RUP role
- Reorganize subsystems (and group) as the project goes on

Next: OO Design or more XP?



- Hamlet and Maybee, chapter 16
- or no reading?

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- New Wiki server
 - <http://pixie.cs.uiuc.edu:8080/SEcourse>
 - Hopefully more reliable, so less admin info here
- HW2 was due, should grade by Nov. 10
 - More feedback on projects from TAs
- Project groups should start meeting TAs
- HW3 is posted, due next Tuesday, Nov. 7
 - State machines and ADTs

Topics on design



- Previously
 - Component-level design
 - Modularity and abstraction
 - Refinement, started

- Today
 - Refinement, finishing
 - XP example (for analysis and design)

Refinement



- Approach to design where you start from a high-level, simple description and then “refine it” with more details
- Some facts from practice
 - Designers bounce between high-level and low-level concerns
 - Designers sometimes redo previous steps
 - Steps often done “out of order”

Refinement myths



You make one pass through the design

Each step adds something to the design

Each step is independent of the others

You iterate because you don't have a methodology

Rational design



“A Rational Design Process: How and Why to Fake It” by David Parnas and P.C. Clemens, *IEEE Transactions on Software Engineering, SE-12:2, February 1986.*

Rational design process - one in which every step has a reason

Top-down design is a myth



- Requirements change
- Requirements are incomplete
- Existence of system changes requirements
- Analysts make mistakes
- Designers make mistakes
- Time reveals better solutions
- Too many decisions to note them all

How and Why to Fake It



- Documentation should be structured so that it describes a sequence of refinements with a reason for each
- Sequence not required to be historical
- Designers can reorganize and rewrite sequence if necessary
- Result: clear design that can be reviewed

Lessons



- Iteration is important part of real process
- Get running code as soon as possible!
- Hide effects of iteration on the documentation
- It is easier to go from design to analysis if the same person does both
- Divide work by subsystem, not by RUP role
- Reorganize subsystems (and group) as the project goes on

Analysis/Design in XP



- Similar to RUP except that
 - Everyone does it
 - Little written, more oral
 - Less is done up-front

Analysis in XP



- Gather user stories from customer
- Make initial object model
- For each user story:
 - Step through user story
 - Note the objects it requires
 - Note the operations it uses
- Clean up the model

Communication with customer



- XP does not require any written requirements documents
- How do you make sure you are building the system the customer wants?
 - Customer is on the team
 - Team writes code only for user stories

User story



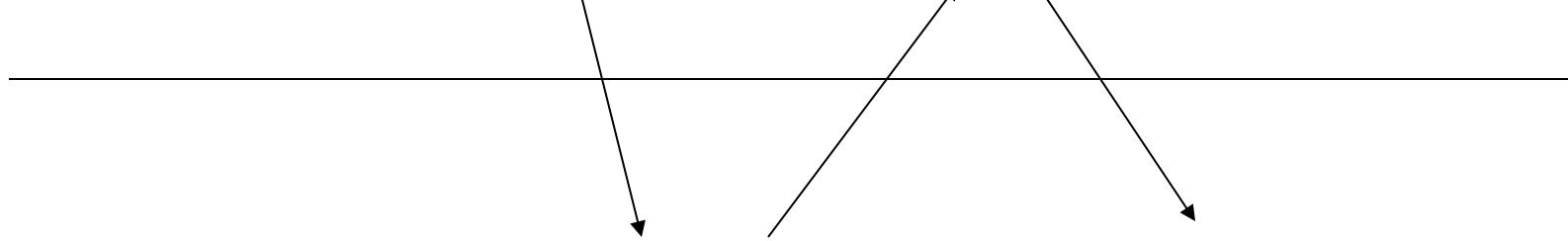
- User story “equivalent” to use case
- User story less detailed than use case
 - Usually just a sentence or two
 - Does not include detailed description
- How do developers know what the customer meant by the user story?

XP timeline (1)

Customer

Write stories

Pick stories



Developers

Estimate stories

Implement stories

XP timeline (2)



■ Startup

- Write stories
- Develop metaphor
- Estimate stories

■ Normal

- Write stories, estimate stories, pick stories
- For each story
 - Write test
 - Write code
 - Refactor

Estimating stories



- Requires requirements
 - Talk to customer
- Requires design
 - Talk to other developers
- Requires experience
 - Work on projects

Who is customer?



- Only one user of system
- Many users of system, all similar
- Many users of system, all different
- Mass-market software

XP does not necessarily



- Develop complete set of user stories
- Develop consistent model of system
- Develop documentation
 - For “gold owner”
 - For users
 - For maintenance programmers

XP requirements



- What “Customer” does
 - Talks with other customers
 - Makes surveys
 - Writes reports, position papers, specs, etc. for management and other customers
 - Writes user stories

XP analysis



- What the developer does
 - Read user stories
 - Talk to each other, play with the design, make sure user story is understood
 - Ask the customer questions
 - Result: estimated user stories, developer who thinks he understands what to do

Objections to XP



- Developer will make too many mistakes
- Too much rework
- Never get around to writing documentation

Strengths of XP



- Lots of communication
- Minimize unnecessary paperwork
- Make at least one customer happy

Modeling example



- Build up model bit by bit
- Look at one requirement at a time
- A new requirement might require
 - Adding to model
 - Changing model
- Model should support all the requirements you've seen so far

Why we model this way



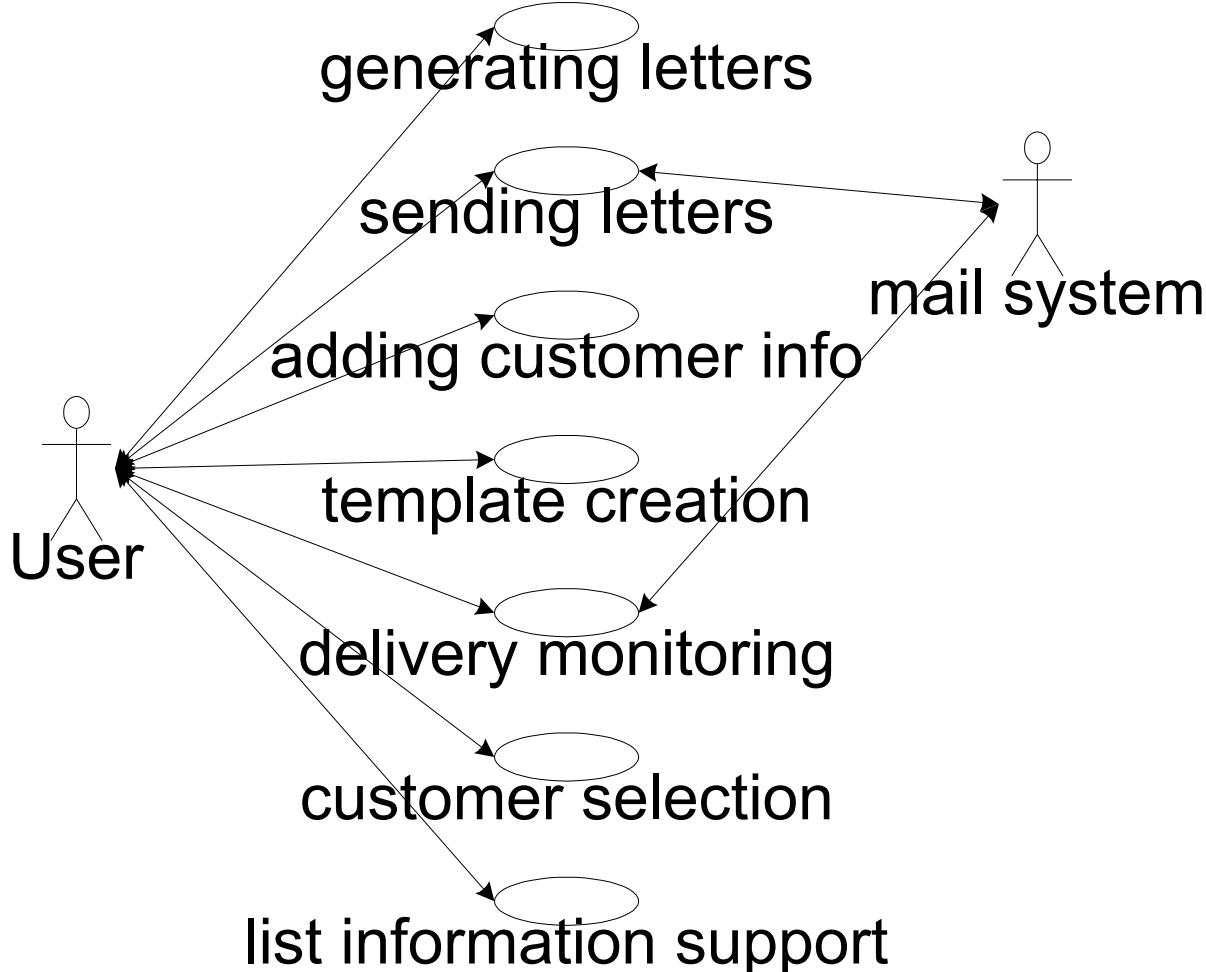
- We can only think about one thing at a time
- Criticism is easier than creation
- As long as we get the right answer, it doesn't matter how we got it

The Viking example



- A direct marketing system
- Sends customized mail and e-mail
- Example from an OOPSLA Design Fest
- Description consists of a set of use cases
- This example is long; we'll cover only a part of it!

Use Case Diagram for The Viking

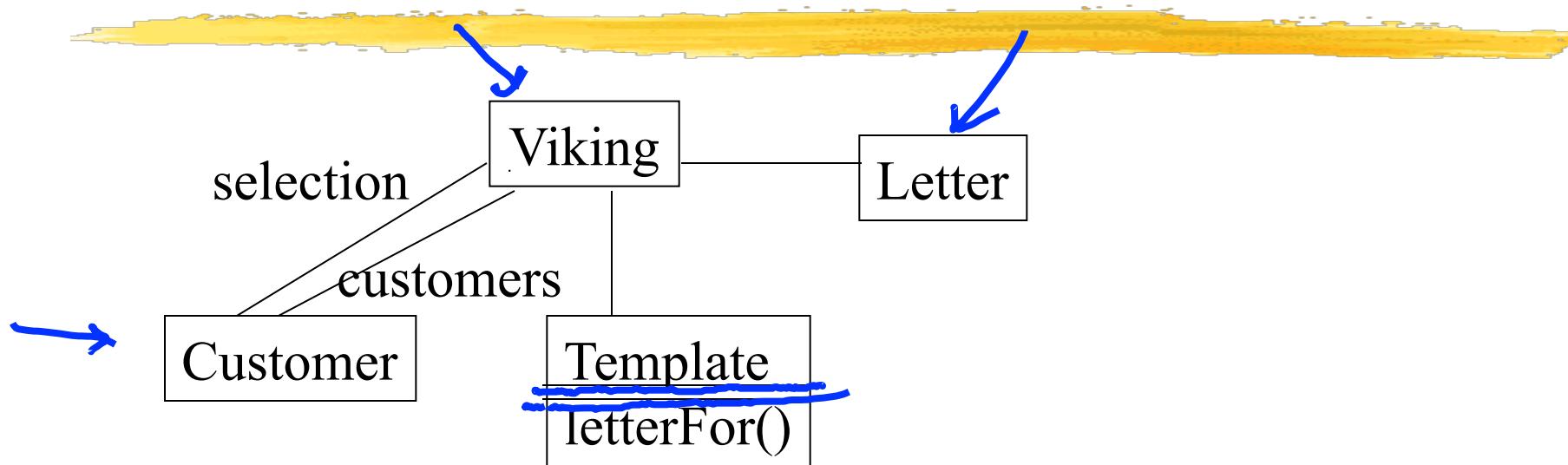


Generating letters



A user selects a set of customers to whom they wish to send letters and a template that defines the letter format. The Viking then generates a letter per customer that is based on filling in the “pluggable” information for the template with customer-related information. The user then previews the result of expanding the template list for each of the customers.

Initial UML model



Template

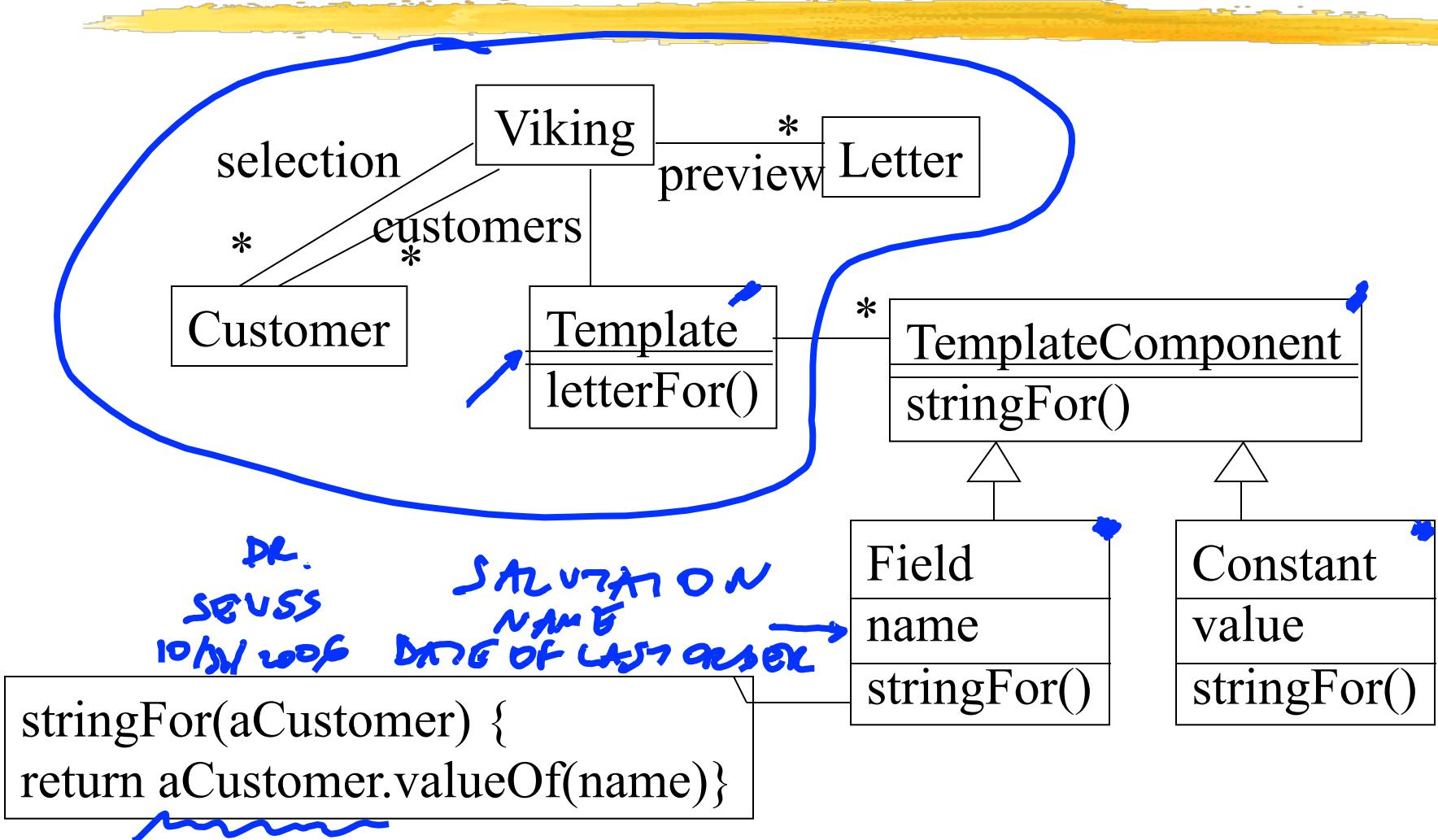


Dear <<Proper Salutation>> <<Customer Name>>
DR. SEVSS,

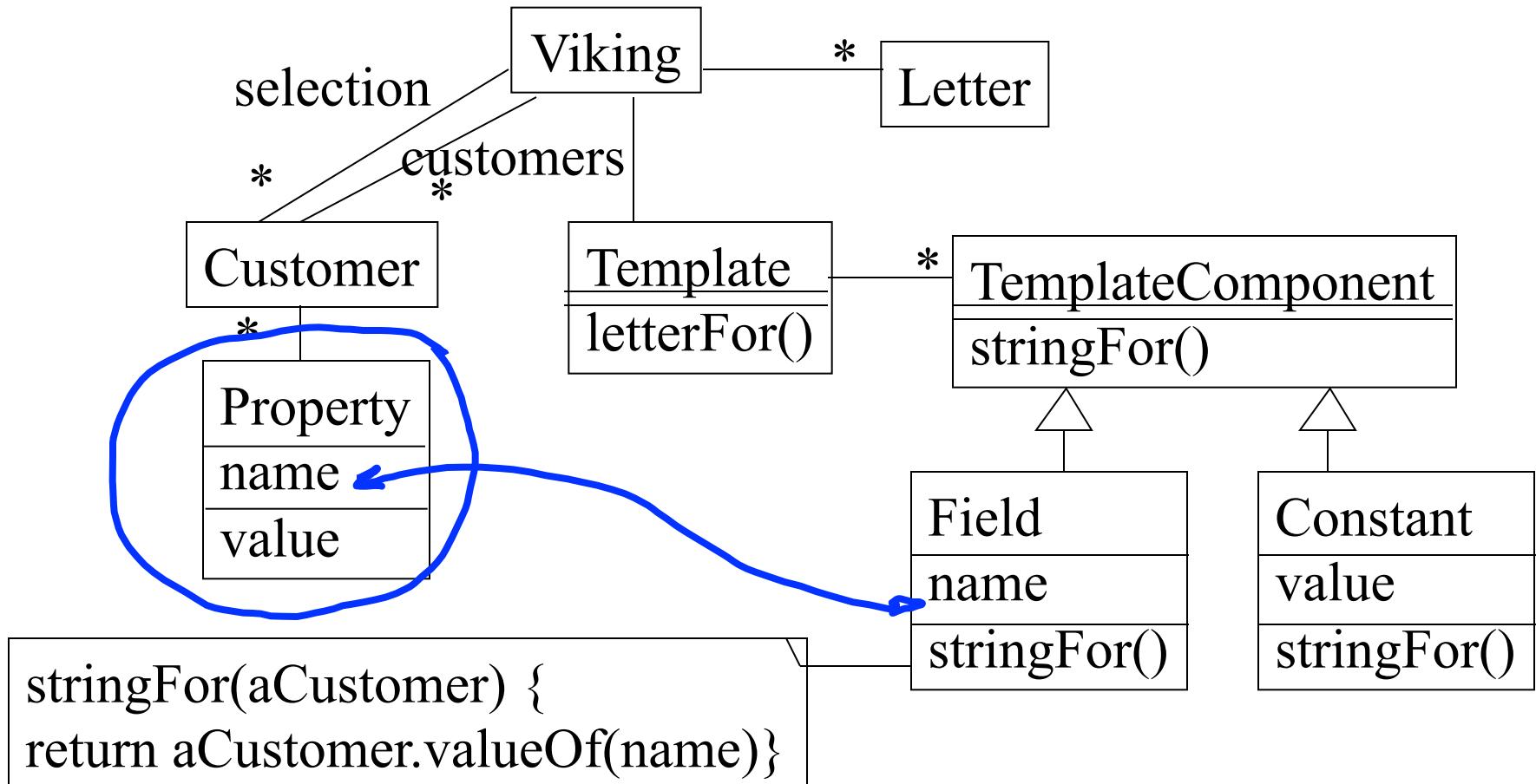
Thank you very much from ordering from us on <<Date of most recent order>>. We recently received several thousand cans of the special ingredient, ...

If you are interested, please click on <http://theHappyViking.com/SpecialOrder/<<special order number>>/Order.html>

Templates have components



Customizable properties



Use template to create a letter for a customer

letter := empty letter; ✓

for each component c of template, add
c.stringFor(customer) to letter

"DEAR"
If c is a constant, c.stringFor() is c.value. ✓

If c is a field, c.stringFor(customer) is
customer.valueOf(c.name)

↳ COMPUTATIONS

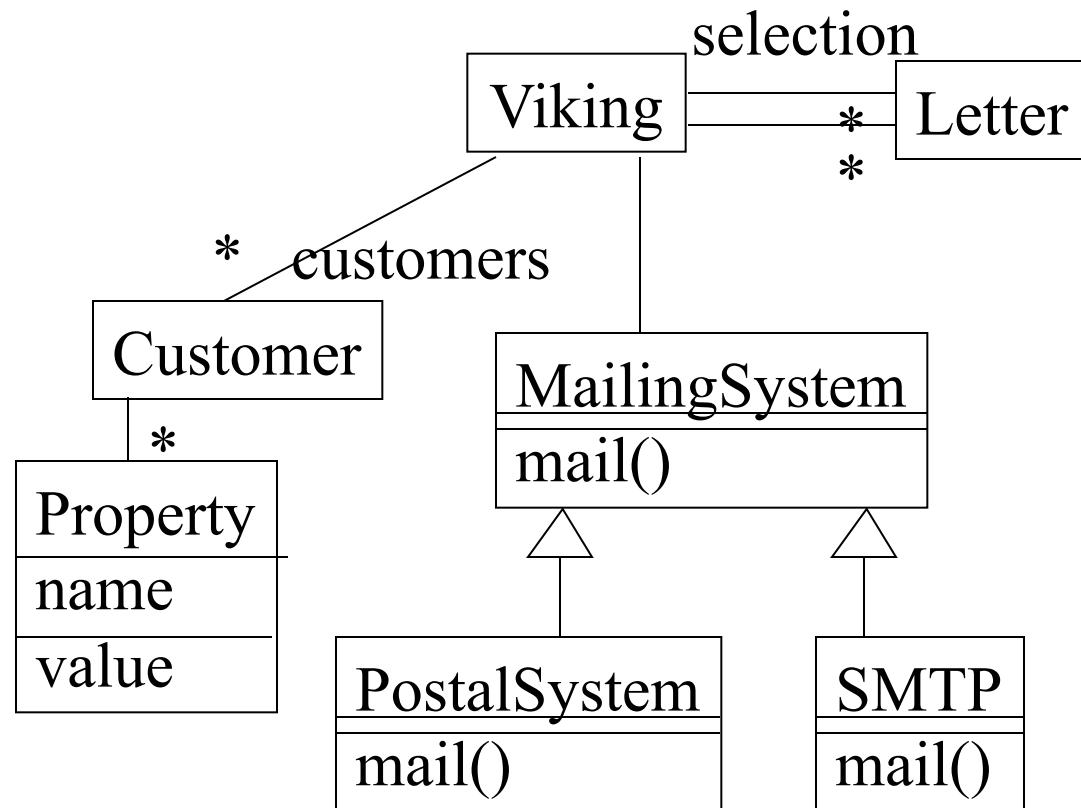
↳ "DEAR"

Sending letters

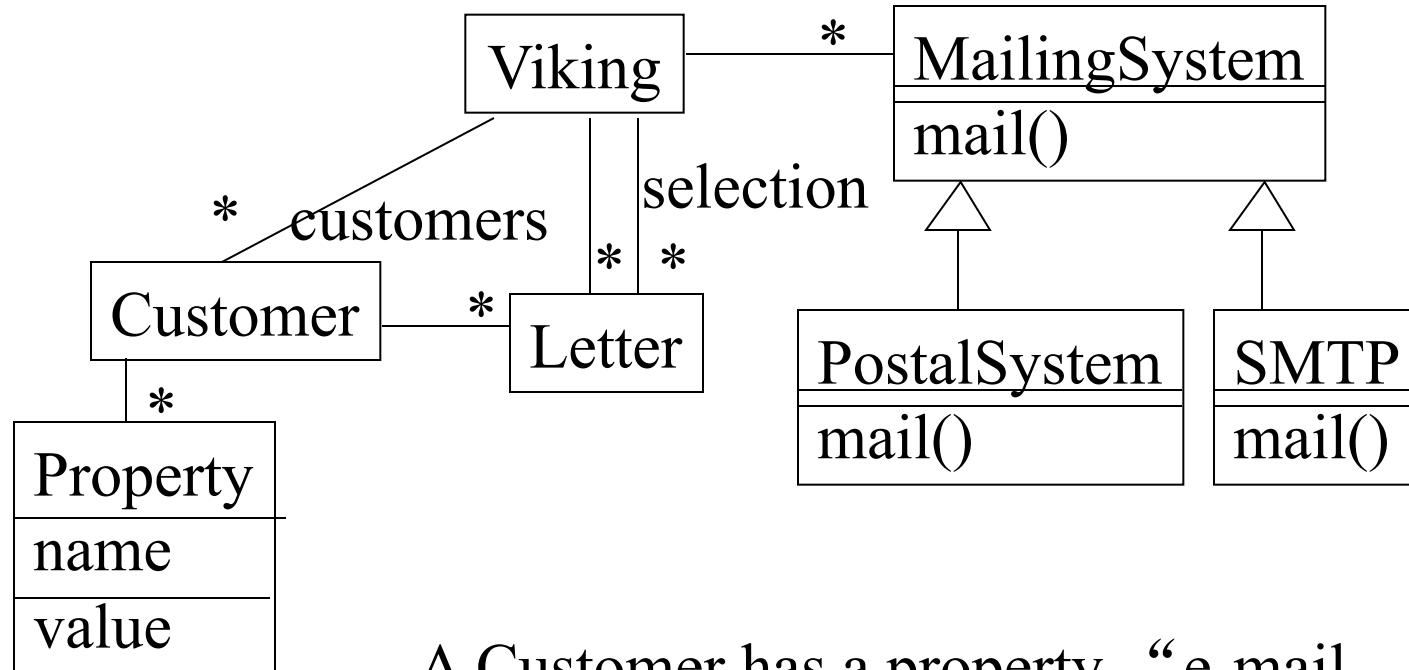


A user chooses from among the generated letters and decides which ones to send out and by which MailingSystem to send them. The Viking should already have information associated with each customer so it can properly distribute the letter by a particular MailingSystem; the user should not need to enter this information as part of the sending process.

One mailing system



Customers have letters



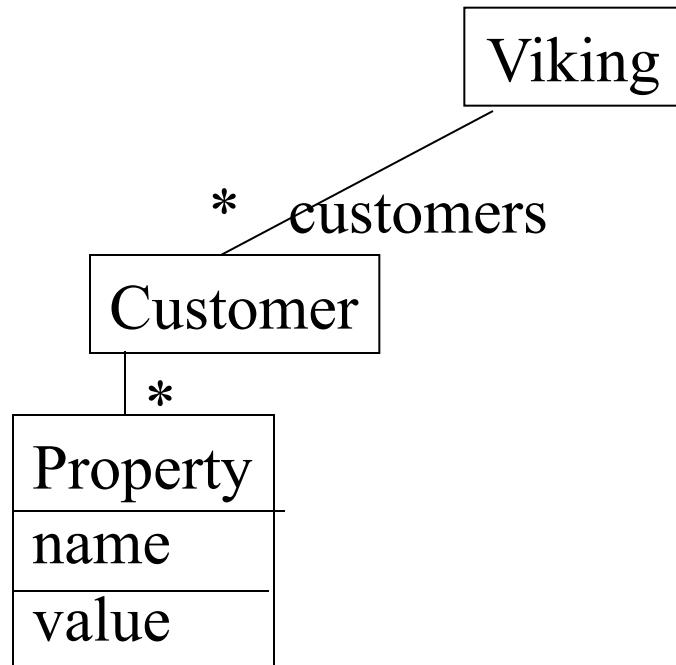
A Customer has a property “e-mail address” and “postal address” which are used by the MailingSystems to obtain an address.

Adding Customer Information



A user knows of a new customer that she wants to add to The Viking. The user can create a new customer entry and record relevant information (Name, Salutation, Address, Recent Purchase, and anything needed by other parts of The Viking) for that customer.

Focus on customers



Interface for creating customer



Create New Customer

Name:

Anna Kurn

Salutation:

Ms.

Address:

712 Maple Ave, Toronto NY

E-mail:

akurn@fipster.com

Recent purchase:

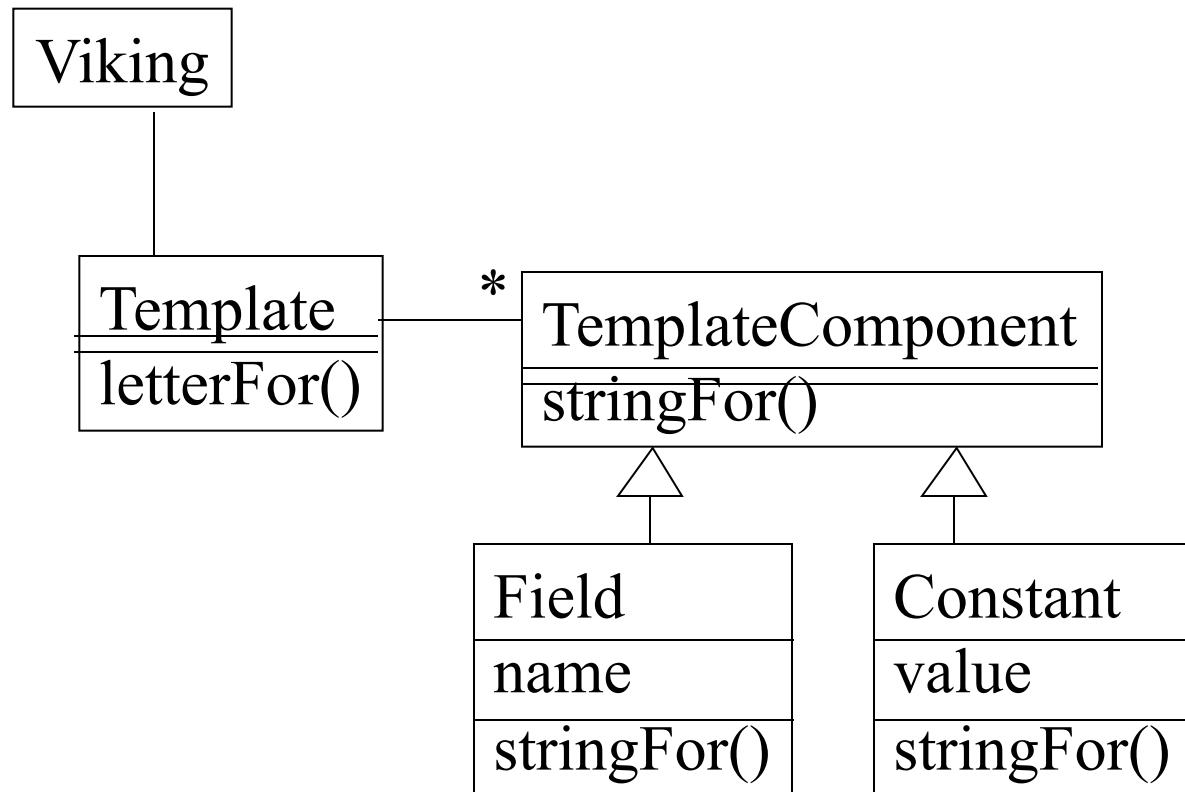
#e45

Template Creation

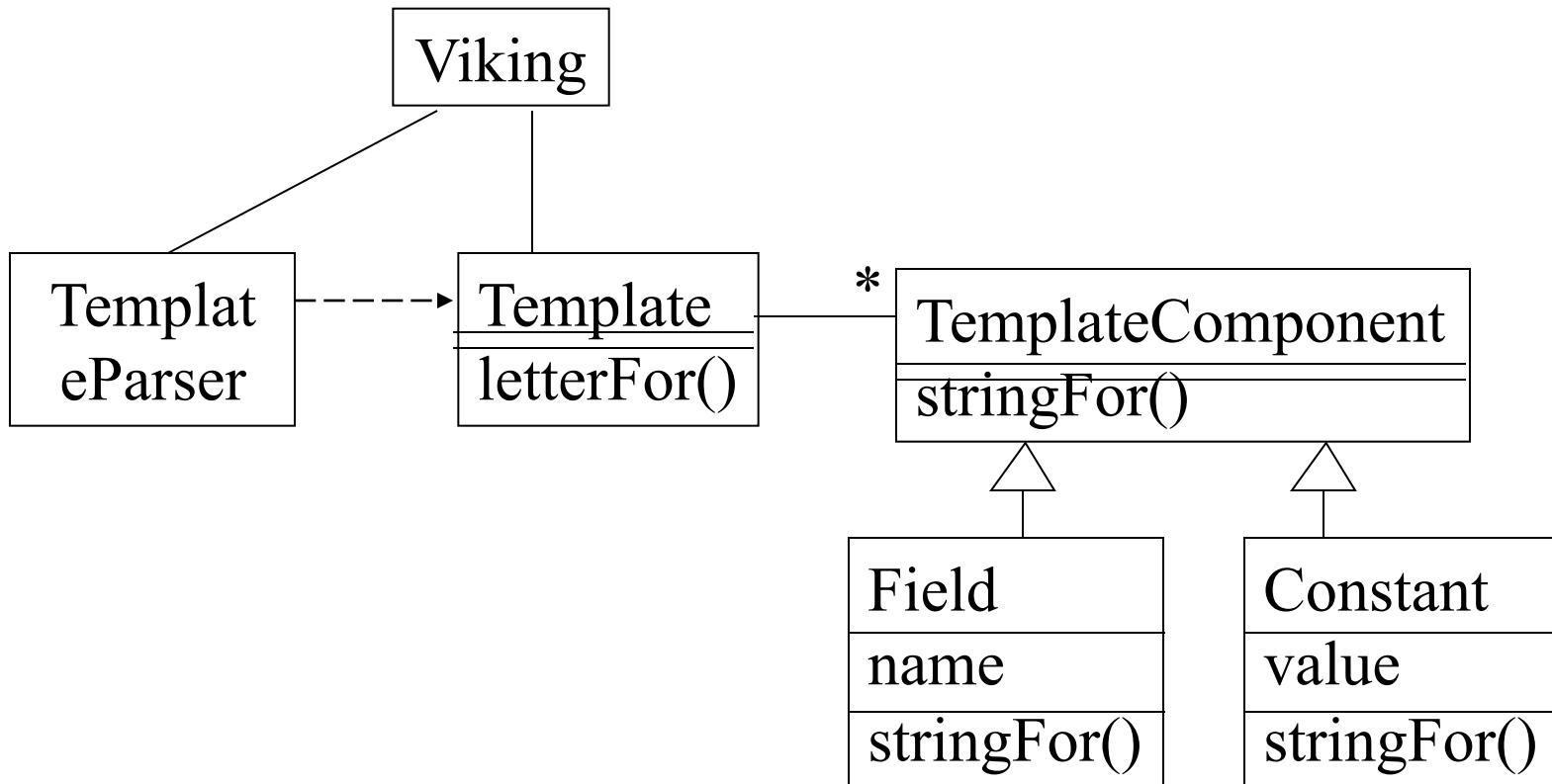


A user creates a new template either from scratch or by copying an existing template. A template needs to support both constant and “pluggable” information, and a user should be able to create a template and preview its appearance.

Focus on templates



Build templates from text



User interfaces



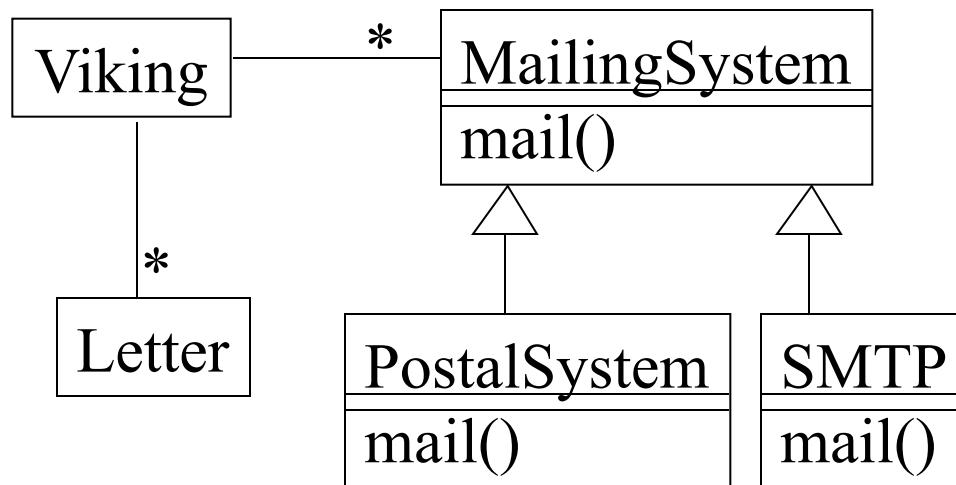
- Generate letters (select a set of customers and a template) and preview them
- Select letters to send and the mailing system to use
- Create new customer and enter info
- Create new template and preview it

Delivery Monitoring

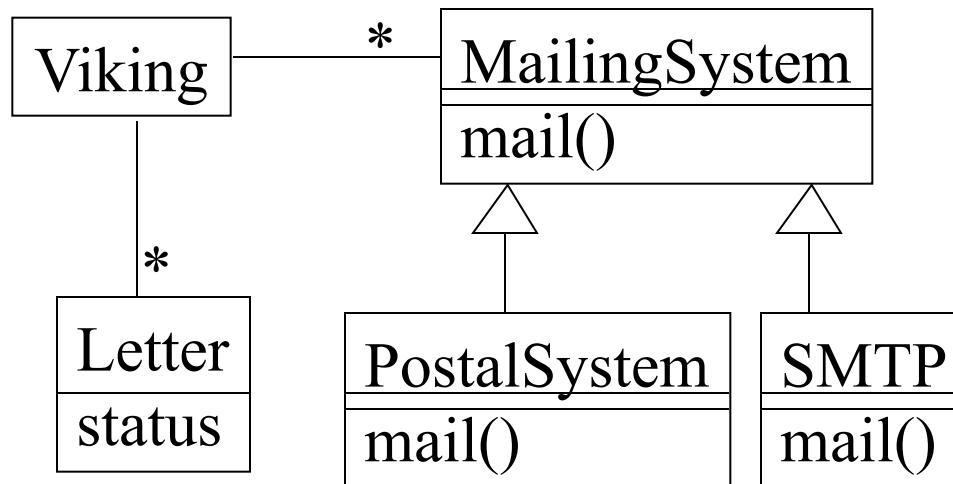


A user reviews the letters to see which have been sent and whether any of them have “failed delivery” . If so, the user can choose to resend them either by the same Mailing System or by a different one. Also, to support this story, the Mailing System must be able to tell The Viking which sent letters “failed delivery” .

Focus on mailing system



Letters have status



Mail delivery failure



- Some failures are permanent (“no such user”) and some are temporary (“mailbox full”). Need different status for each.
- Can have secondary addresses.
- “Select permanent failures and send to secondary address” is special case of “select messages and send them”
- Need UI for changing status

Customer selection by criteria



Upgrade the template from Story#1 to support selecting the set of customers by various search criteria. For example, select all customers who spent more than US\$100 on their most recent order, or all customers who have every bought a particular product.

Consdierations



- What query language?
 - SQL?
 - Dialog box?
- What criteria?
 - Any history?
 - Orders, products
- Change the UI for “select customers”

Conclusion



- Only partly done
 - Process to finish is the same
 - Each step makes progress
 - Keep track of open issues and make sure they get resolved
 - Amount you write down depends on how much you remember
- projects

Hints for XP



- Pair programming
 - Schedule time (put in plan)
 - Meet together as much as possible
- Testing – use Junit or similar
 - <http://www.junit.org>
 - <http://www.xprogramming.com/software.htm>
- Refactoring – see <http://refactoring.com>

Next: OO Design



- Hamlet and Maybee, chapter 16

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- New Wiki server
 - <http://pixie.cs.uiuc.edu:8080/SEcourse>
 - Please report broken links (such as for HW3)
- HW3 is due on Tuesday, Nov. 7
 - State machines and ADTs
- Project groups should start meeting TAs
 - Get feedback on projects from TAs
- Office hours: Mon 3-4, Tue 10-11, Thu 2-3

Topics on design



- Previously
 - Component-level design
 - Modularity and abstraction
 - Refinement
 - XP example (for analysis and design)
- Today
 - OO design (and some RUP reminders)
 - Time permitting: revisit Viking example

Architecture



Early design decisions

- | High-level
- | Pervasive
- | Expensive to change

Architectural styles

Architecture is used to:



- Decide what design problems to work on first
- Divide the system into modules
 - Divide the developers into teams
 - Subcontract work to other groups
- Decide what technology to use

Object-oriented design



- How to view objects/classes in your software system
- Book describes three methods
 - Booch
 - Evolved into UML/RUP
 - Schlaer-Meller
 - Object information, object state, processing/actions
 - Responsibility-driven design
 - Objects and their collaborations

Responsibility-driven design



- Object model is made up of cards
- Each card represents a class
- Class lists
 - Name, superclass
 - Responsibilities
 - Collaborators

Responsibility-driven design



- Start with a “scenario” – a use case
- Add to classes/responsibilities/ collaborations to implement scenario
- Only change classes because of scenario
- Complaint? Invent a scenario!

Responsibility-driven design



- Like RUP and XP
 - Use case driven
 - Incremental
- Like XP
 - Lousy documentation
 - Best seen live

OOD



■ What is OOD good for?

- Business systems
- GUIs
- Editor

■ What is it bad for?

- Real-time control
- Compilers?

What order?



- Use cases first?
- Processes first?
- Data model first?
- GUI first?

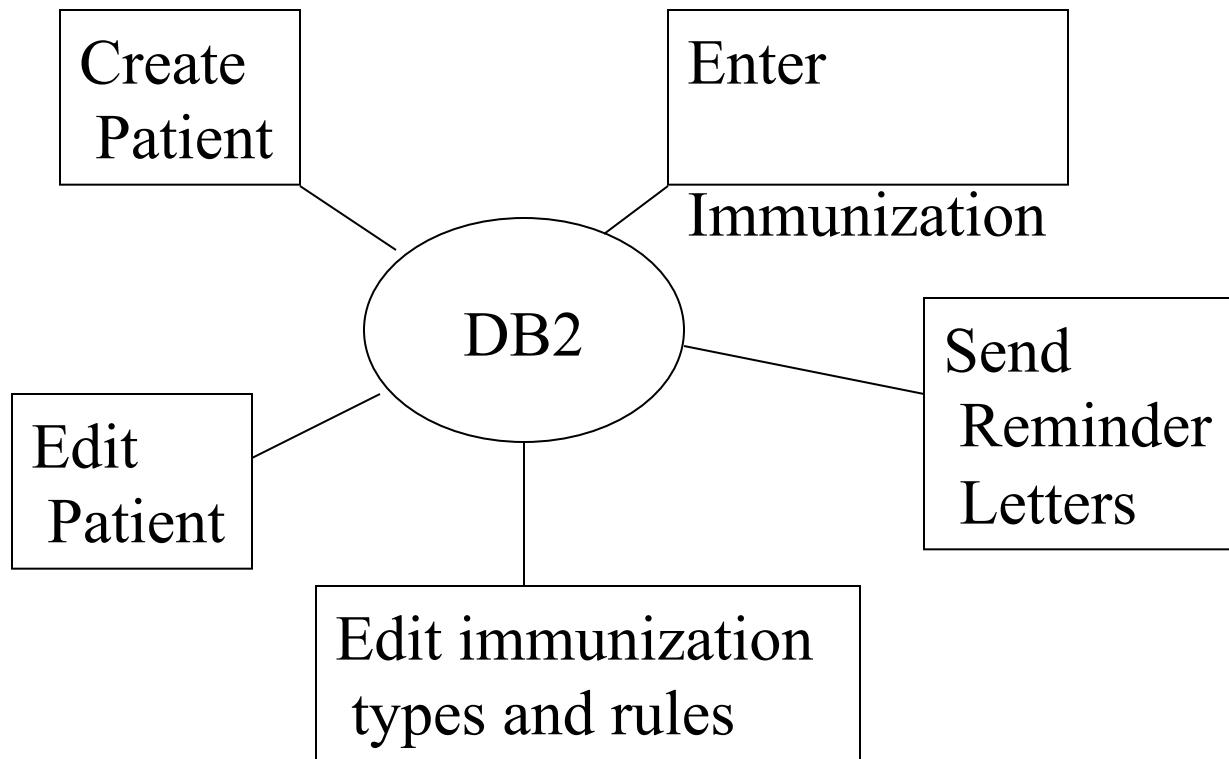
- Depends on architectural style

Data-centered architecture



- Database design is key
- Not hard to add data, can be hard to change
- Behavior less important, easy to change
- RUP can be overkill
- XP can be a problem because refactoring a RDBMS schema can be hard

Data-centered architecture

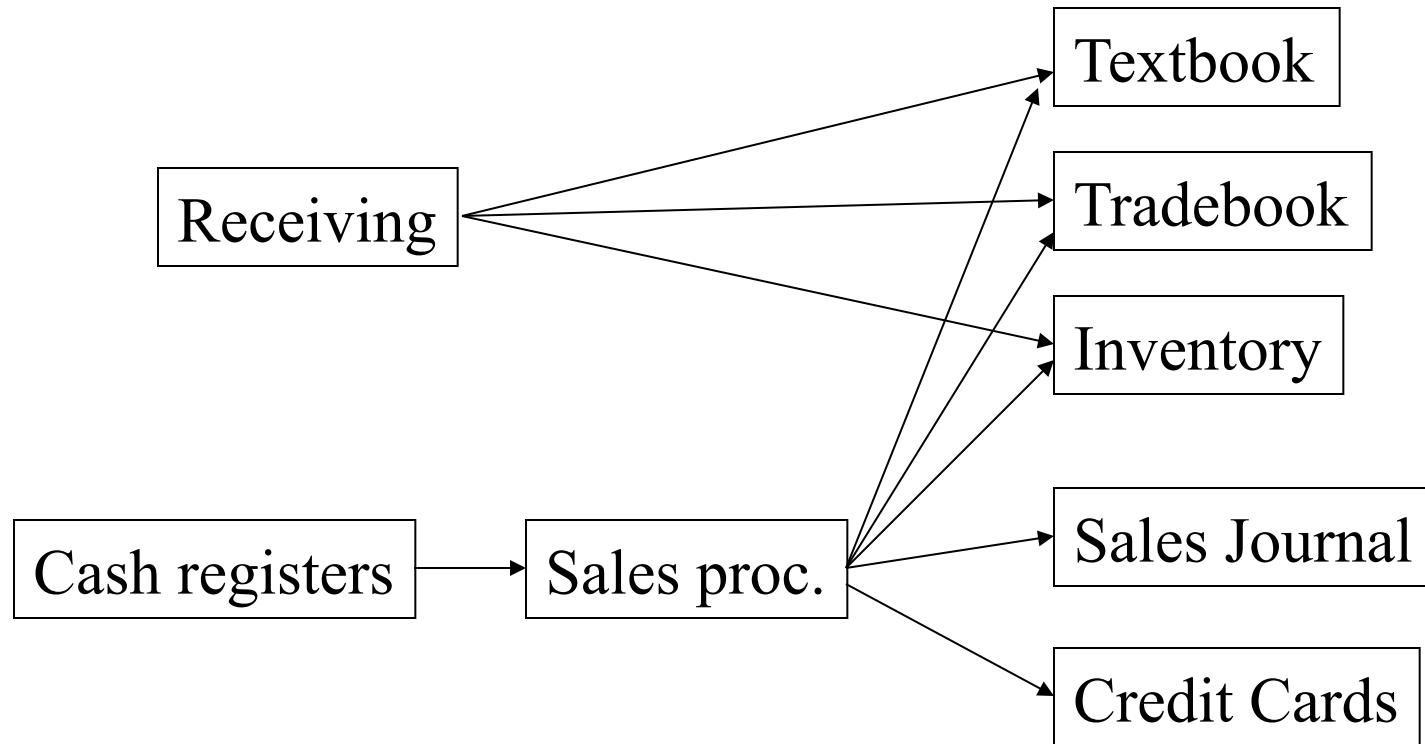


Transaction-oriented



- Transactions are stored in a database
 - Timecards and paychecks (payroll)
 - Purchases and sales (retail)
 - Telephone calls (telephone billing)
- System keeps track of totals
- Totals are function of transactions
- Rules for computing totals tend to be complex and frequently change

Transactions

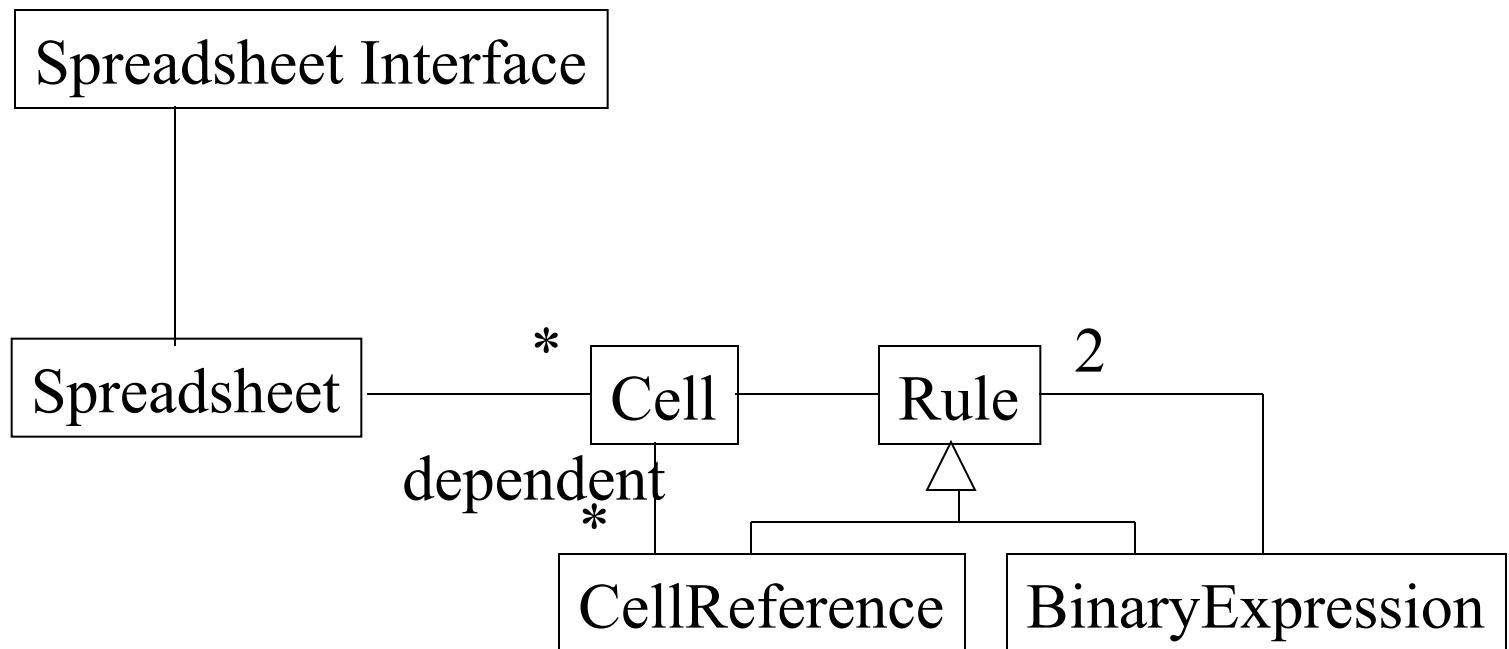


Work-piece



- User is manipulating complex data
- Data only changes when user changes it
- Model of what user is thinking, not of the world
- User interface is often important
- Efficiency usually is not important
- Ideal for object-oriented programming

Work-piece



Realtime-control



- System must respond in a short time to events in the real world
- Must be efficient enough
- Data usually trivial, GUI usually simple
- Usually contains feedback loops

Trade-offs



- How do you pick an architectural style?
 - Depends on what you know
 - Depends on architectural qualities that are important
- What is important?
 - Flexibility and ease of change
 - Efficiency
 - Reliability

Quality Attributes



- Desired properties of entire system, not particular features
- Non-functional requirements
- Usability, maintainability, flexibility, understandability, reliability, efficiency, ...

ATAM



- Architectural Trade-off Analysis Method
 - Collect scenarios (architectural use cases)
 - Collect requirements, constraints, and environment descriptions
 - Describe candidate architectural styles
 - Evaluate quality attributes
 - Identify sensitivity of quality attributes
 - Critique candidate architectures

Trade-offs



- How it is really done
 - Experts pick a way that seems best and start to use it
 - If problems arise, they reconsider
-
- A paper on general decision-making
<http://www.fastcompany.com/online/38/klein.html>

What is an architect?



- Responsible for choosing architecture
- Responsible for communicating with client
- Responsible for making trade-offs about features
- Chief builder - “Architect also implements”
- Leader of developers

Your projects



- More than one person can play the architect role
- More than one person MUST implement parts of the code
 - But not everyone has to implement code
- Document your process
 - TAs will provide more detailed grading criteria

Next: Quality Assurance



- Hamlet and Maybee, Chapter 19 on Testing
(not 17 or 18)

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- HW2 graded, get hard copies from Ganesh
 - Grades posted on Wiki
- New Wiki server
 - <http://pixie.cs.uiuc.edu:8080/SEcourse>
 - Consider switching to another Wiki (for CS428)
- HW3 postponed for Thursday, Nov. 9
 - State machines and ADTs
- Project groups start/keep meeting with TAs
 - Get feedback on projects from TAs

Topics



- Covered design
 - Component-level design
 - Modularity and abstraction, refinement
 - XP example (for analysis and design)
 - OO design (and some RUP reminders)
- Today
 - ADTs: one more example (others in Lecture 17)
 - Software Quality Assurance

Example ADT



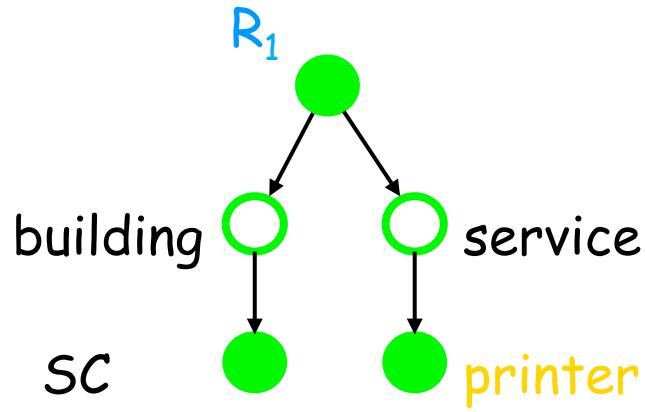
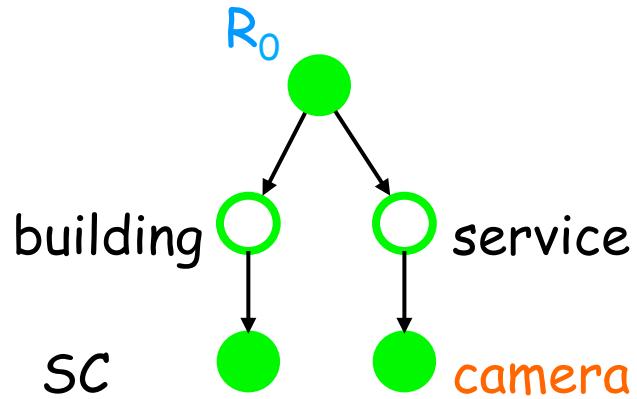
- Intentional Naming System (INS)
 - We'll look at this one
- Phone book
 - You can see it with more details elsewhere
 - <http://www.mit.edu/~6.170/lectures/adts.pdf>
- Something from your projects
 - If you have questions, you can ask now or through emails/newsgroup

INS



- [Adjie-Winoto et al., SOSP 1999]
- Resource discovery in dynamic networks
- Names in INS mean what not where
 - Traditional: ‘128.174.252.214’ or ‘ds3203.cs.uiuc.edu’
 - Intentional: ‘color printer for transparencies’
- Intentional names
 - Hierarchy of attributes and values

Intentional names

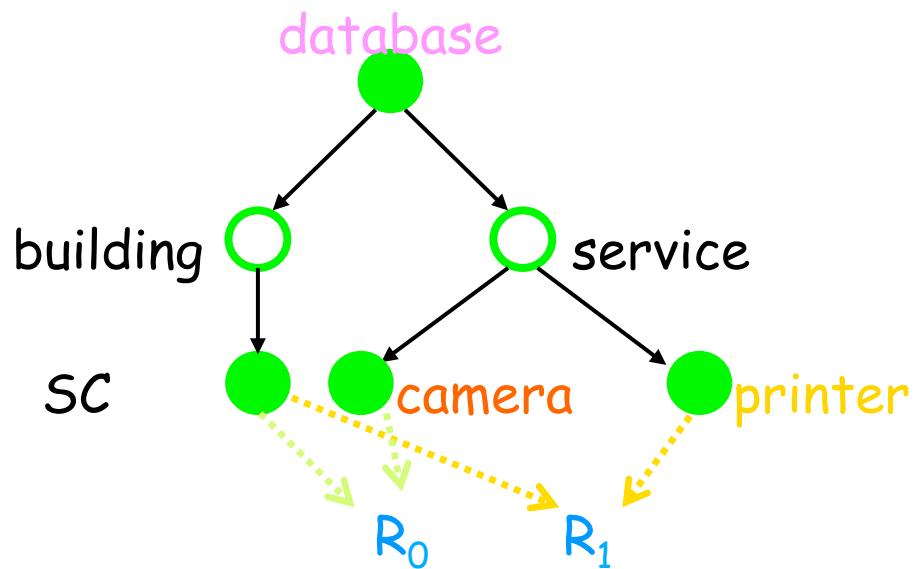
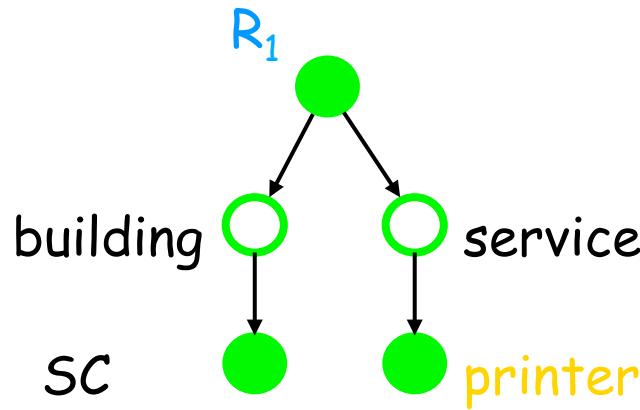
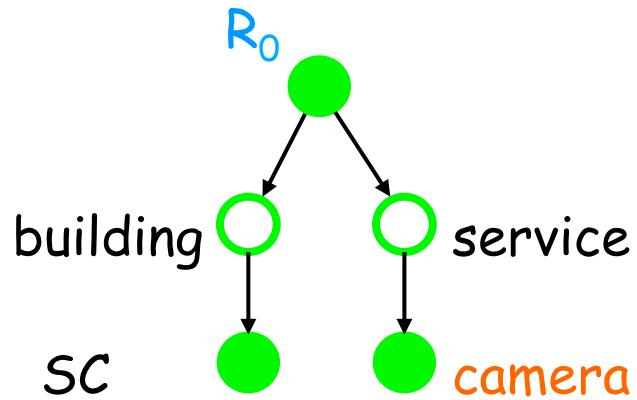


Databases in INS

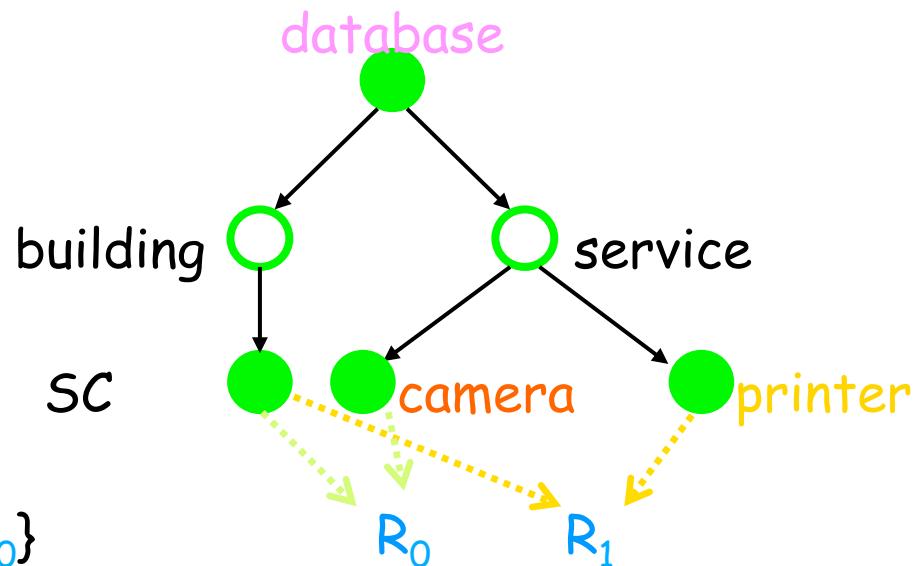
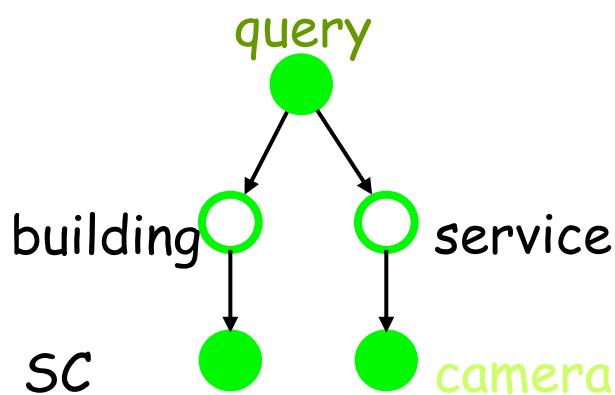
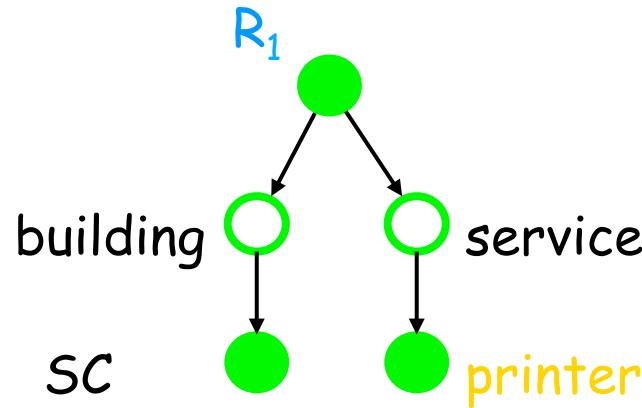
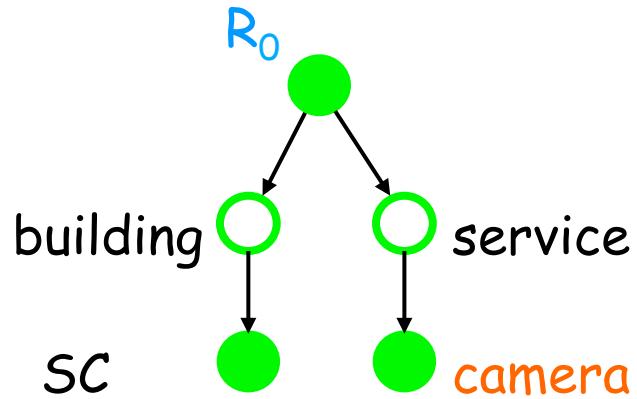


- Collect information about resources
- Respond to queries

Add to database



Lookup into database



$\text{lookup}(\text{query}, \text{database}) = \{R_0\}$

ADT for database



- Suppose that we've already designed
 - Intentional Name (this is not that easy)
 - Resource (this is trivial)
- We want to design database
 - Values?
 - Operations?
 - Properties?

Operations



- Constructors (types?)

- empty:
 - add:

- Observers (types?)

- lookup:
 - get:

Axioms



- Notation: OO (receiver) or math (functions)
 - Basic axioms for collections (preconditions?)
-
- Main property: subtree matching in lookup

Design vs. implementation



- Implementation for INS had a few hundreds lines of Java code
 - Hard to understand high-level picture
 - Had bugs
 - Cannot fit in one slide
- Axioms can fit into one slide
 - Easier(?) to understand
 - Can analyze automatically (found some bugs)

Software quality assurance



- SQA: not just testing
- How can you tell if software has high quality?
- How can we measure the quality of software?
- How can we make sure software has high quality?

Perspective on quality



■ Customer

- System not crashes
- System follows documentation
- System is logical and easy to use

■ Developer

- System is easy to change
- System is easy to understand
- System is pleasant to work on

SQA



■ Potential mistakes

- Quality is conformance to requirements and standards
- Variation control is the heart of quality control (mass production unlike software)

■ Iterative view

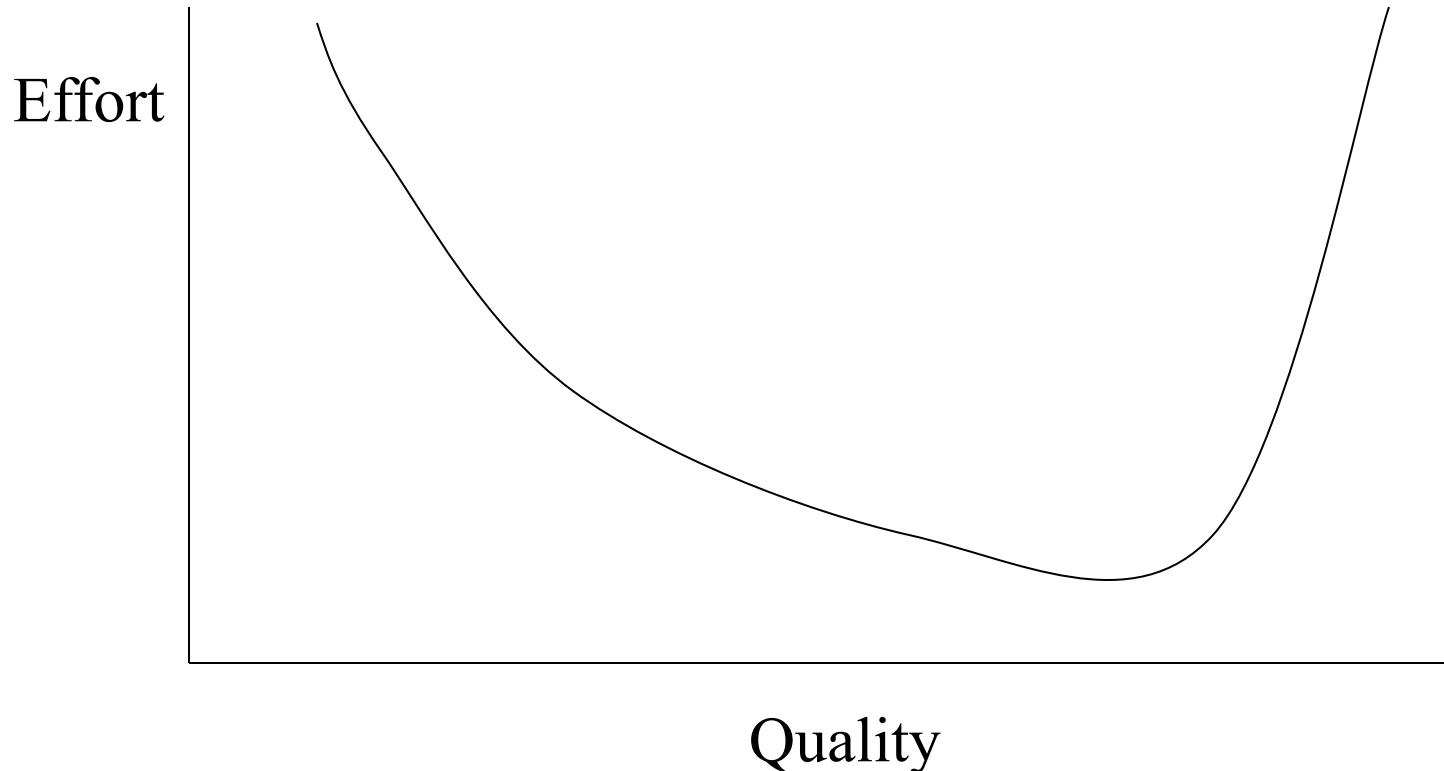
- Feedback and continual improvement is the real heart of quality software

Total quality management

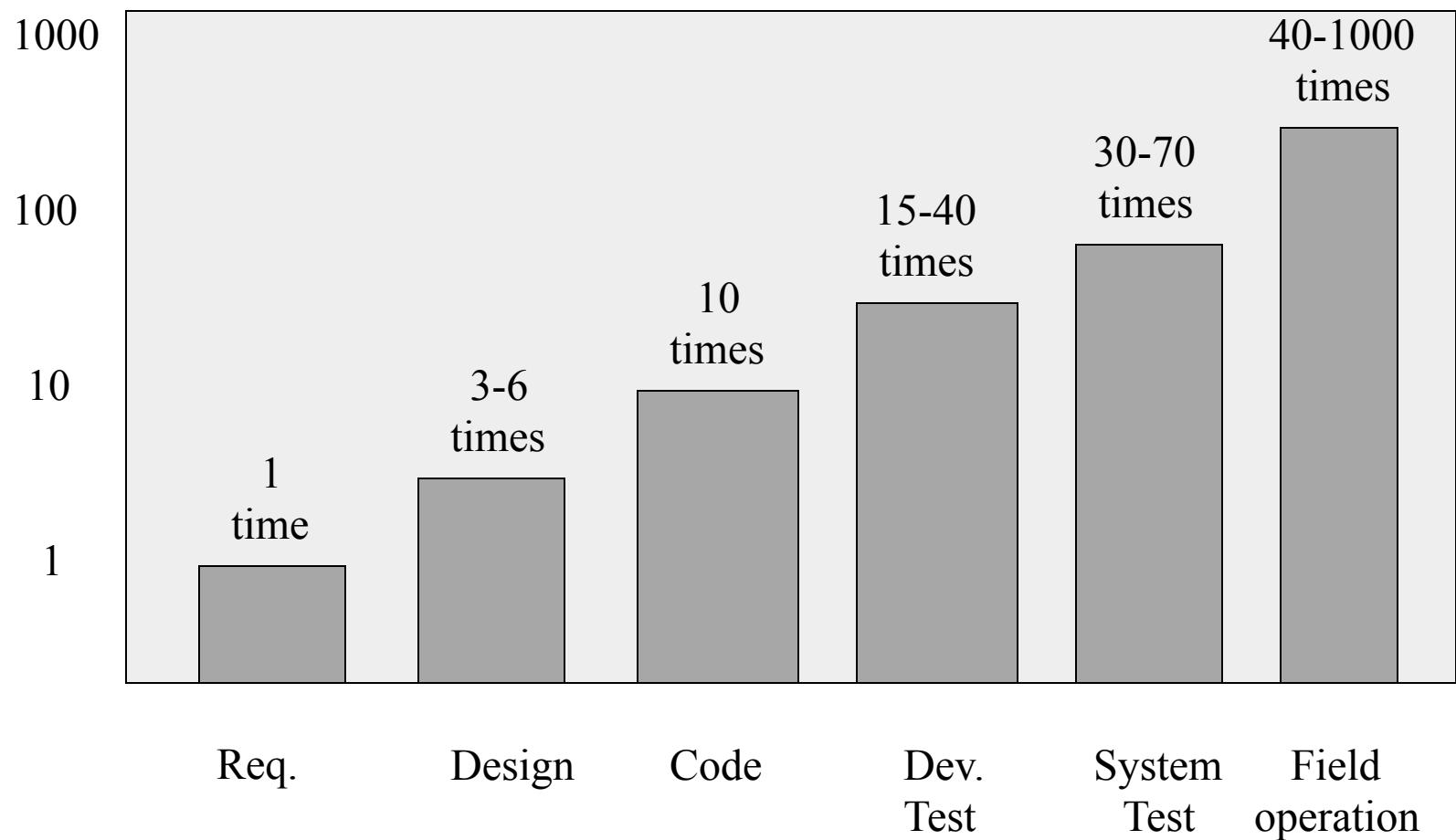


- Factories
- Goal is for every item coming off the assembly line to be perfect
- Management, production, engineering, QA
- Everyone is involved in quality
- Develop a reliable, repeatable process
- Continuously improve the process

“Quality is free”



Cost of fixing an error



Error: Terminology?



- Anomaly
- Bug
- Crash
- Defect
- Error
- Failure/fault
- ...

Failure vs. flaw



- Failure - program didn't work right
- Flaw - mistake in the text of the program
- Failure analysis (debugging) - what flaw caused this failure?
- Flaw analysis - what is wrong with our process that allowed this flaw to be created and not detected?

Failure costs



- Internal
 - Rework
 - Repair
 - Failure analysis
- External
 - Resolving complaints
 - Returning and replacing product
 - Help line

Prevention costs



- Prevention
 - Planning
 - Managing and collecting information
 - Reviews
- Appraisal
 - Inspection
 - Testing

Johnson's law



“If you don’t test for it, your system
doesn’t have it.”

Is it easy to use?

Easy to maintain?

Does it crash?

Does it match the documentation?

Does it make customers happy?

Ways not to improve quality



- Say “Be more careful!”
- Say “Quality is important.”
- Find out whose fault it is and fire him

How to improve quality



- Measure and compare
- Determine root cause of problems
- Create ways to eliminate problems

Metrics



- If you don't see it, it doesn't exist
- Measure quality over time (metrics)
- Display in a public place

- Make quality goals, then check to see if you meet them

How to appraise quality



- Requirements
 - Reviews by customers
 - Prototyping
- Analysis and design models
 - Formal reviews, inspections
- Current system
 - Bug reports
 - User tests
 - Surveys

Bug tracking



- Keep track of
 - Who reported the bug (the failure)
 - Description of the failure
 - Severity
 - The flaw that caused this failure
 - Who is repairing it
 - The repair

Bug tracking



- Use information about failures to estimate reliability
- Compare
 - Critical nature of failure
 - Iteration failure discovered
 - Module that had the flaw

Use quality information to make decisions



- “Must repair all level 1 failures before shipping”
- “Half of all level 1 and 2 failures in the alpha release were in the Call Processing module; we should rewrite it.”
- “Half of all level 1 and 2 defects found in the design reviews were in Call Processing; we should rewrite it.”

Bug tracking



- Discover the flaw (defect) that caused each bug
- Categorize flaws
- Look at categories with the most flaws and improve your process to eliminate them

Technical reviews



- A way to evaluate the quality of requirements, designs, and software
- A way to improve the quality of requirements, designs, and software
- A way to educate new developers and ensure that developers are consistent
- *Proven to be cost-effective!*

Main goal: Evaluate quality



- Produce a report describing
 - Potential problems
 - Summary of overall quality
 - Pass/fail
- Evaluated by expert outsiders
 - Must know enough
 - Shouldn't know too much

Secondary goal: Improve quality



- Find flaws
- Enforce standards
- Improve standards
- Provide feedback to management

The review team



- Leader (moderator)
- Recorder
- Reviewers

Leader



- Responsible for obtaining a good review - or reporting why a good review wasn't possible
- Good review - one that accurately describes the quality of the product
- Make sure that reviewers have all the material they need for the review
- Get a time and place for the review

Recorder



- Responsible to provide information for an accurate report of the review
- Typically writes notes on a “flip chart” or other public medium
- At end of review, recorder gives summary and makes sure the team agrees
- Recorder helps leader make final report

Reviewers



- Study product in advance and take notes
- Have a check-list of review criteria
- Give both positive and negative comments
- Raise issues, don't resolve them
- Must be technically competent
- Stick to standards - or stick the standards

Result of review



■ Review summary

- Who, what, when and the conclusion

■ Issues list

- Can result in more detailed reports
- Give priority to issues
- Can be disagreement on issues
- Most issues are about product, but can also be about process or standards

Walkthrough



- Producer guides reviewers through the product
- Easier on reviewers
 - Can cover more material
 - More reviewers can participate
- Good for training the reviewers
- Not as good for evaluating and improving the product

Inspection



- Confine attention to a few aspects of the product, one at a time
- Less preparation by reviewers than for review
- More preparation by leader - must give detailed instructions to reviewers

Reviewing products



- Must compare each product with the products that it is based on
 - Class diagram based on use cases
 - Code in RUP based on design
 - Code in XP based on tests
- Some products not based on any other products
 - User stories?
 - Vision statement?

Checklist: Smalltalk code



- Class names make sense
- There is a good class comment that defines variables
- Classes not too big (less than 20 methods)
- Classes have behavior
- Method names
 - Spelled out in full
 - Accessors are “attribute” and “attribute:”
 - Methods with side effects are commands
 - Methods without side effects are adjectives or noun phrases
- Methods not too big (less than 15 lines)

Summary



- Reviews are an important quality control technique
- Pair-programming is a kind of informal review
- Use reviews to improve your process, not just your product

Next: Continuing on SQA



- Hamlet and Maybee, Chapter 19 on Testing
(not 17 or 18)

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- HW2 graded, get hard copies from Ganesh
 - Grades posted on Wiki
- New Wiki server
 - <http://pixie.cs.uiuc.edu:8080/SEcourse>
 - Consider switching to another Wiki (for CS428)
- HW3 postponed for Thursday, Nov. 9
 - State machines and ADTs
- Project groups start/keep meeting with TAs
 - Get feedback on projects from TAs

Topics



- Covered design
 - Component-level design
 - Modularity and abstraction, refinement
 - XP example (for analysis and design)
 - OO design (and some RUP reminders)
- Today
 - ADTs: one more example (others in Lecture 17)
 - Software Quality Assurance

Example ADT



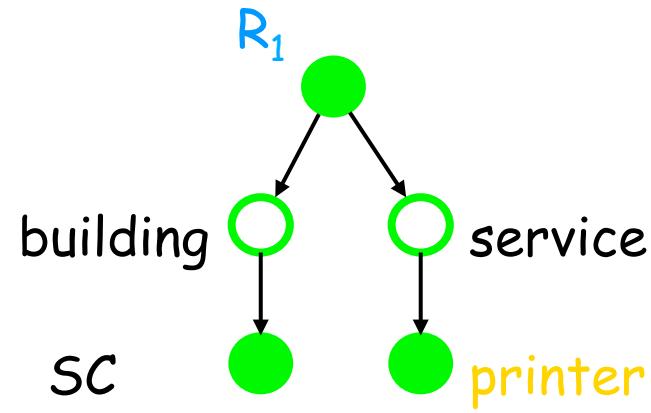
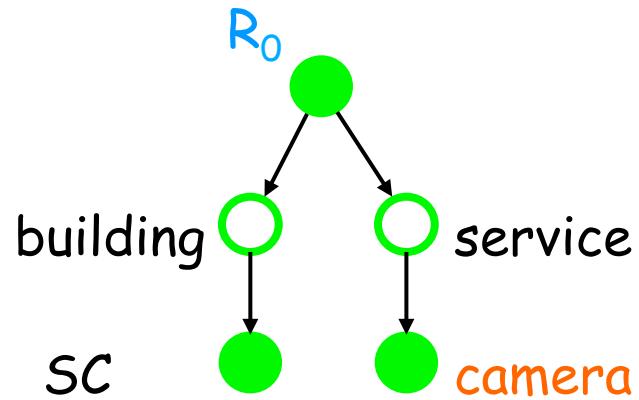
- Intentional Naming System (INS)
 - We'll look at this one
- Phone book
 - You can see it with more details elsewhere
 - <http://www.mit.edu/~6.170/lectures/adts.pdf>
- Something from your projects
 - If you have questions, you can ask now or through emails/newsgroup

INS



- [Adjie-Winoto et al., SOSP 1999]
- Resource discovery in dynamic networks
- Names in INS mean what not where
 - Traditional: ‘128.174.252.214’ or ‘ds3203.cs.uiuc.edu’
 - Intentional: ‘color printer for transparencies’
- Intentional names
 - Hierarchy of attributes and values

Intentional names

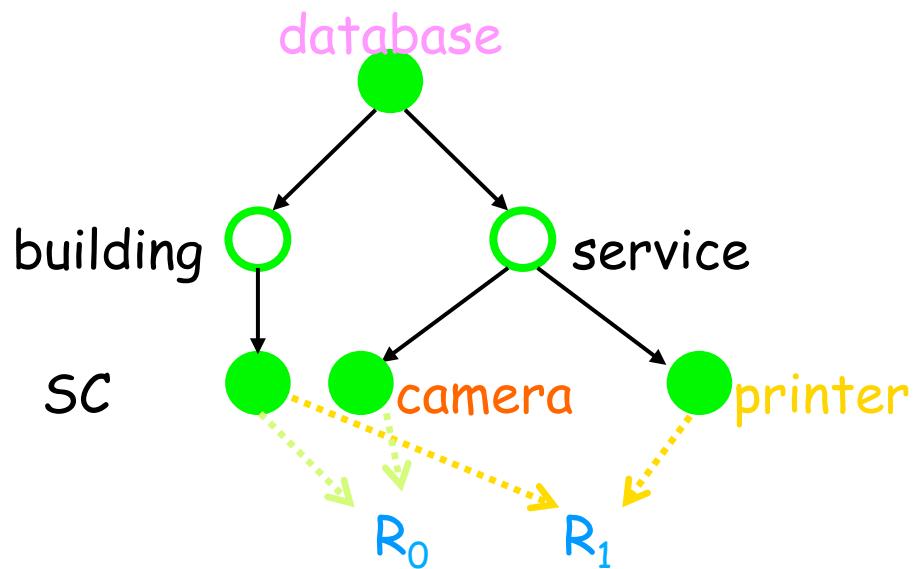
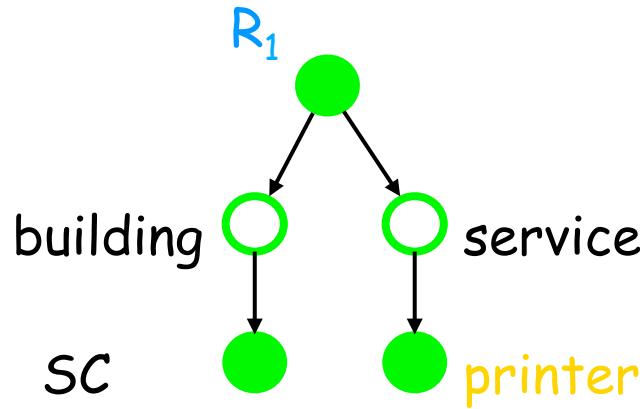
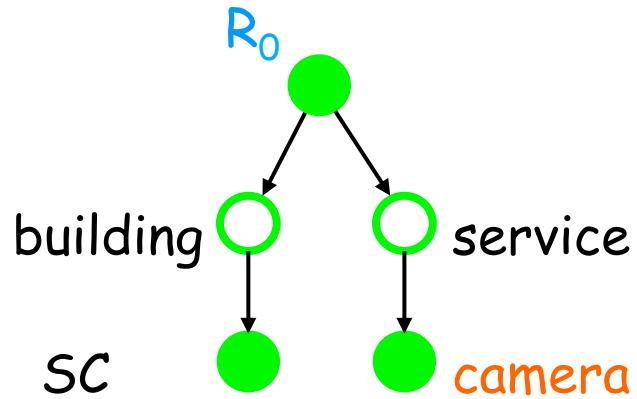


Databases in INS

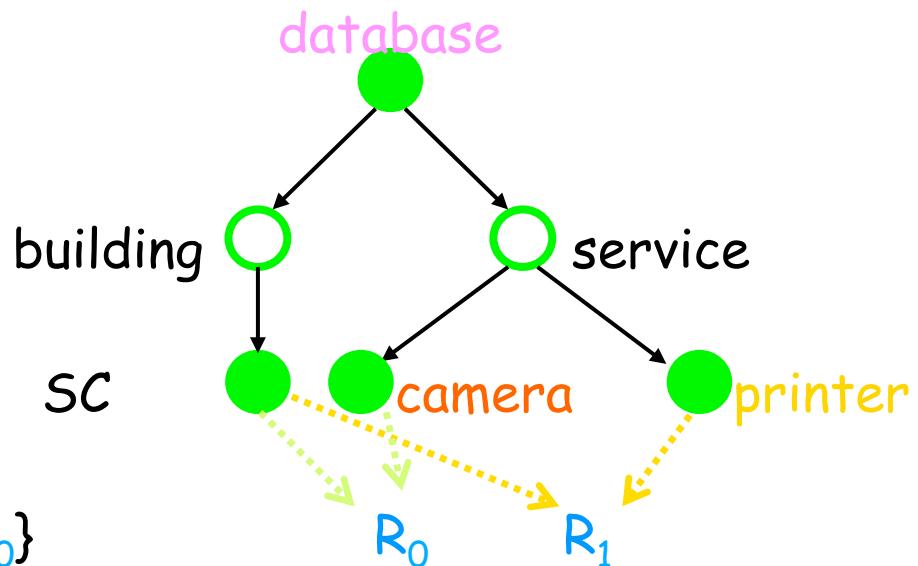
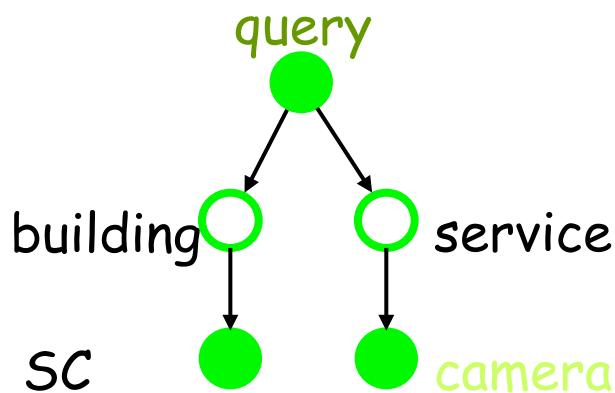
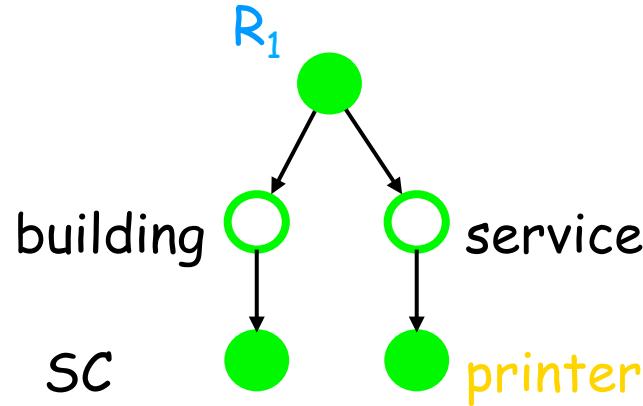
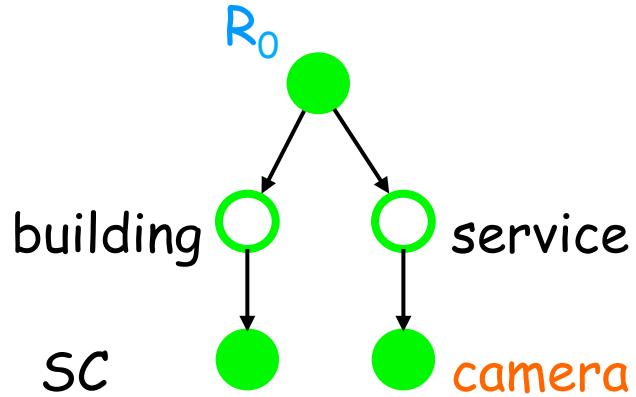


- Collect information about resources
- Respond to queries

Add to database



Lookup into database



$\text{lookup}(\text{query}, \text{database}) = \{R_0\}$

ADT for database



- Suppose that we've already designed
 - Intentional Name (this is not that easy)
 - Resource (this is trivial)
- We want to design database
 - Values?
 - Operations?
 - Properties?

Operations



- Constructors (types?)

- empty:
 - add:

- Observers (types?)

- lookup:
 - get:

Axioms



- Notation: OO (receiver) or math (functions)
 - Basic axioms for collections (preconditions?)
-
- Main property: subtree matching in lookup

Design vs. implementation



- Implementation for INS had a few hundreds lines of Java code
 - Hard to understand high-level picture
 - Had bugs
 - Cannot fit in one slide
- Axioms can fit into one slide
 - Easier(?) to understand
 - Can analyze automatically (found some bugs)

Software quality assurance



- SQA: not just testing
- How can you tell if software has high quality?
- How can we measure the quality of software?
- How can we make sure software has high quality?

Perspective on quality



■ Customer

- System not crashes
- System follows documentation
- System is logical and easy to use

■ Developer

- System is easy to change
- System is easy to understand
- System is pleasant to work on

SQA



■ Potential mistakes

- Quality is conformance to requirements and standards
- Variation control is the heart of quality control (mass production unlike software)

■ Iterative view

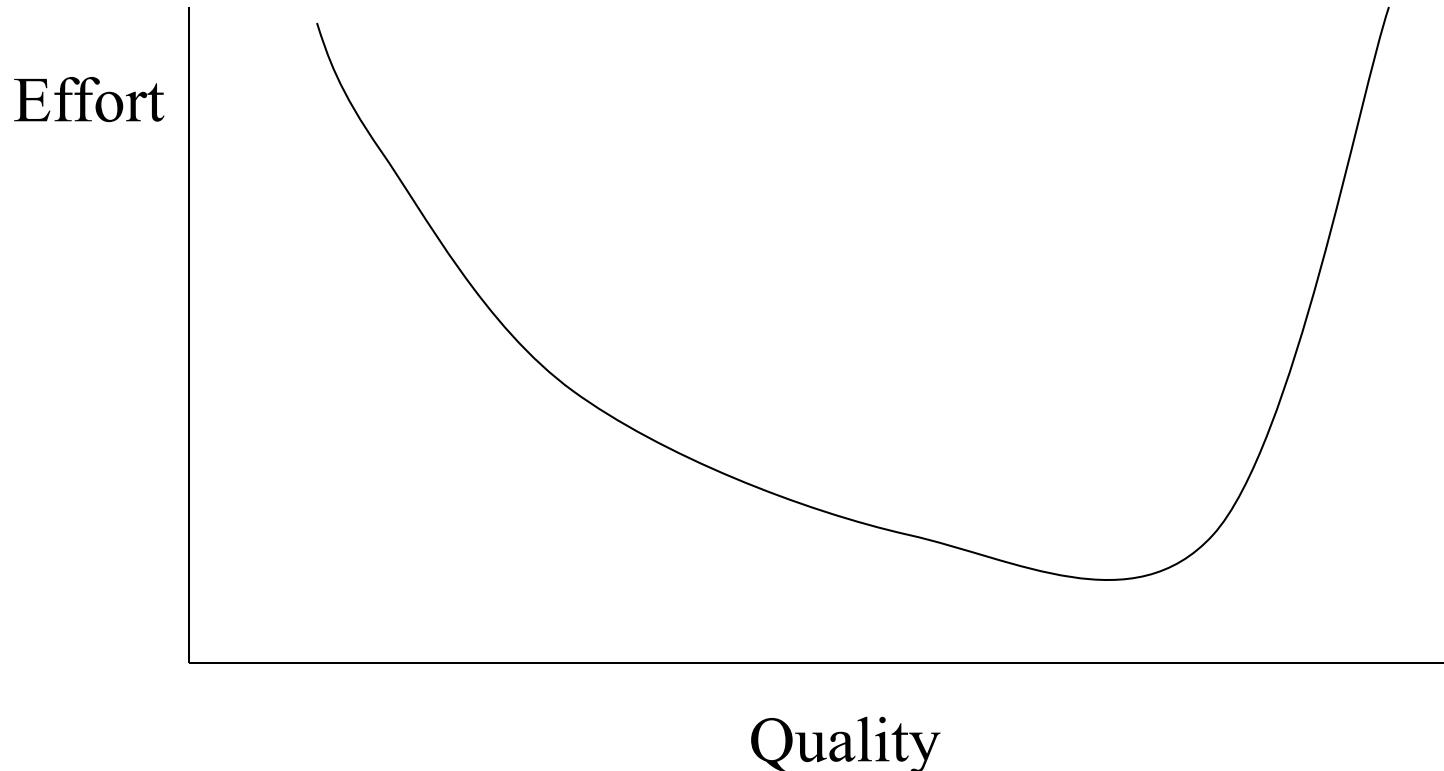
- Feedback and continual improvement is the real heart of quality software

Total quality management

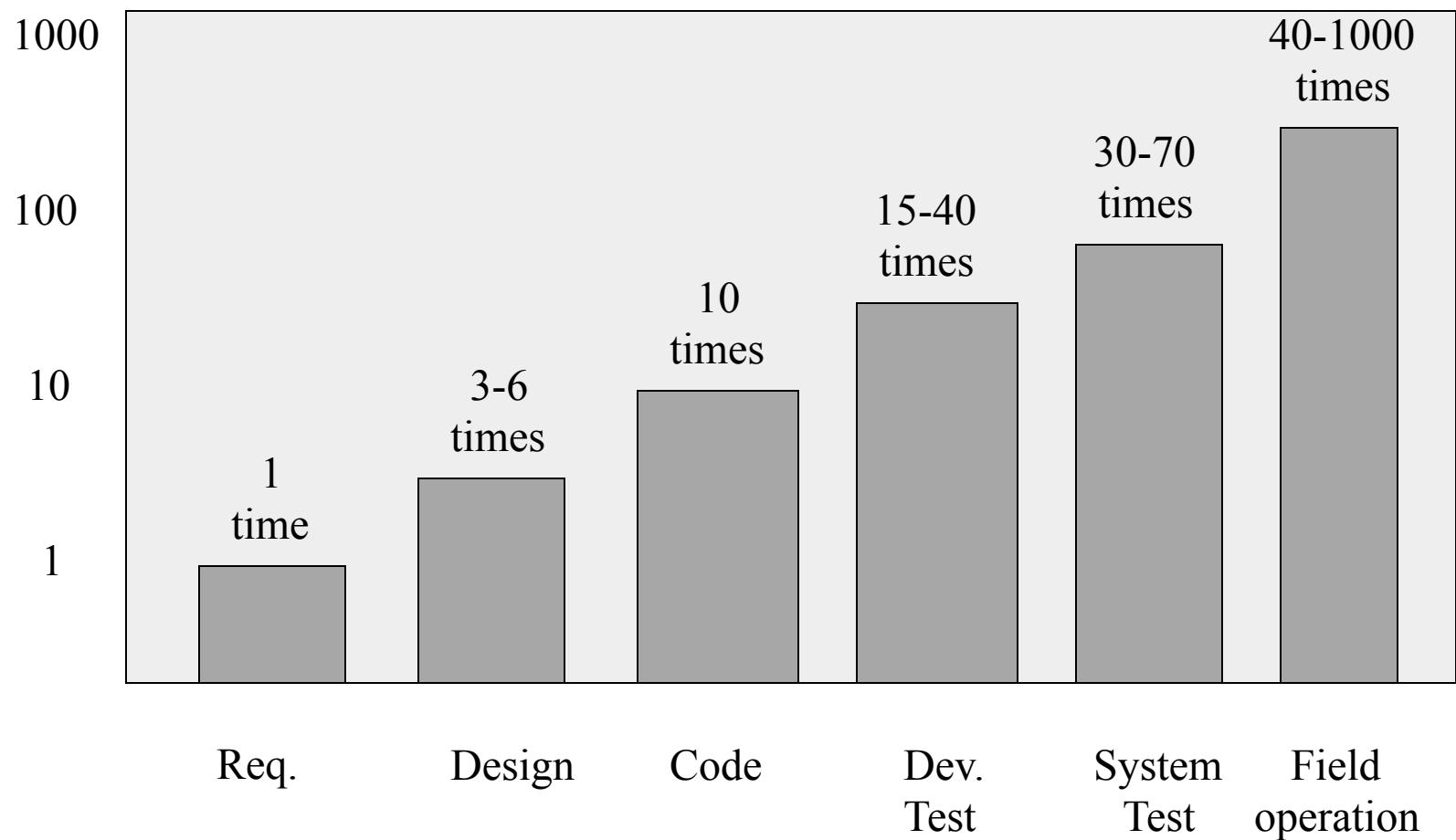


- Factories
- Goal is for every item coming off the assembly line to be perfect
- Management, production, engineering, QA
- Everyone is involved in quality
- Develop a reliable, repeatable process
- Continuously improve the process

“Quality is free”



Cost of fixing an error



Error: Terminology?



- Anomaly
- Bug
- Crash
- Defect
- Error
- Failure/fault
- ...

Failure vs. flaw



- Failure - program didn't work right
- Flaw - mistake in the text of the program
- Failure analysis (debugging) - what flaw caused this failure?
- Flaw analysis - what is wrong with our process that allowed this flaw to be created and not detected?

Failure costs



- Internal
 - Rework
 - Repair
 - Failure analysis
- External
 - Resolving complaints
 - Returning and replacing product
 - Help line

Prevention costs



- Prevention
 - Planning
 - Managing and collecting information
 - Reviews
- Appraisal
 - Inspection
 - Testing

Johnson's law



“If you don’t test for it, your system
doesn’t have it.”

Is it easy to use?

Easy to maintain?

Does it crash?

Does it match the documentation?

Does it make customers happy?

Ways not to improve quality



- Say “Be more careful!”
- Say “Quality is important.”
- Find out whose fault it is and fire him

How to improve quality



- Measure and compare
- Determine root cause of problems
- Create ways to eliminate problems

Metrics



- If you don't see it, it doesn't exist
- Measure quality over time (metrics)
- Display in a public place

- Make quality goals, then check to see if you meet them

How to appraise quality



- Requirements
 - Reviews by customers
 - Prototyping
- Analysis and design models
 - Formal reviews, inspections
- Current system
 - Bug reports
 - User tests
 - Surveys

Bug tracking



- Keep track of
 - Who reported the bug (the failure)
 - Description of the failure
 - Severity
 - The flaw that caused this failure
 - Who is repairing it
 - The repair

Bug tracking



- Use information about failures to estimate reliability
- Compare
 - Critical nature of failure
 - Iteration failure discovered
 - Module that had the flaw

Use quality information to make decisions



- “Must repair all level 1 failures before shipping”
- “Half of all level 1 and 2 failures in the alpha release were in the Call Processing module; we should rewrite it.”
- “Half of all level 1 and 2 defects found in the design reviews were in Call Processing; we should rewrite it.”

Bug tracking



- Discover the flaw (defect) that caused each bug
- Categorize flaws
- Look at categories with the most flaws and improve your process to eliminate them

Technical reviews



- A way to evaluate the quality of requirements, designs, and software
- A way to improve the quality of requirements, designs, and software
- A way to educate new developers and ensure that developers are consistent
- *Proven to be cost-effective!*

Main goal: Evaluate quality



- Produce a report describing
 - Potential problems
 - Summary of overall quality
 - Pass/fail
- Evaluated by expert outsiders
 - Must know enough
 - Shouldn't know too much

Secondary goal: Improve quality



- Find flaws
- Enforce standards
- Improve standards
- Provide feedback to management

The review team



- Leader (moderator)
- Recorder
- Reviewers

Leader



- Responsible for obtaining a good review - or reporting why a good review wasn't possible
- Good review - one that accurately describes the quality of the product
- Make sure that reviewers have all the material they need for the review
- Get a time and place for the review

Recorder



- Responsible to provide information for an accurate report of the review
- Typically writes notes on a “flip chart” or other public medium
- At end of review, recorder gives summary and makes sure the team agrees
- Recorder helps leader make final report

Reviewers



- Study product in advance and take notes
- Have a check-list of review criteria
- Give both positive and negative comments
- Raise issues, don't resolve them
- Must be technically competent
- Stick to standards - or stick the standards

Result of review



■ Review summary

- Who, what, when and the conclusion

■ Issues list

- Can result in more detailed reports
- Give priority to issues
- Can be disagreement on issues
- Most issues are about product, but can also be about process or standards

Walkthrough



- Producer guides reviewers through the product
- Easier on reviewers
 - Can cover more material
 - More reviewers can participate
- Good for training the reviewers
- Not as good for evaluating and improving the product

Inspection



- Confine attention to a few aspects of the product, one at a time
- Less preparation by reviewers than for review
- More preparation by leader - must give detailed instructions to reviewers

Reviewing products



- Must compare each product with the products that it is based on
 - Class diagram based on use cases
 - Code in RUP based on design
 - Code in XP based on tests
- Some products not based on any other products
 - User stories?
 - Vision statement?

Checklist: Smalltalk code



- Class names make sense
- There is a good class comment that defines variables
- Classes not too big (less than 20 methods)
- Classes have behavior
- Method names
 - Spelled out in full
 - Accessors are “attribute” and “attribute:”
 - Methods with side effects are commands
 - Methods without side effects are adjectives or noun phrases
- Methods not too big (less than 15 lines)

Summary



- Reviews are an important quality control technique
- Pair-programming is a kind of informal review
- Use reviews to improve your process, not just your product

Next: Continuing on SQA



- Hamlet and Maybee, Chapter 19 on Testing
(not 17 or 18)

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- HW2 graded, get hard copies from Ganesh
 - Grades posted on Wiki
- HW3 due today
 - Will grade it before the Thanksgiving break
- HW4 posted, due next Thursday, Nov. 16
 - Reviews done in pairs across different groups
 - Create “pairs” of “pairs” as soon as possible
- Project groups start/keep meeting with TAs
 - Get feedback on projects from TAs

Software quality assurance



- SQA: not just testing
- How can you tell if software has high quality?
- How can we measure the quality of software?
- How can we increase the quality of software?

Perspective on quality



■ Customer

- System not crashes
- System follows documentation
- System is logical and easy to use

■ Developer

- System is easy to change
- System is easy to understand
- System is pleasant to work on

SQA



■ Potential mistakes

- Quality is conformance to requirements and standards
- Variation control is the heart of quality control (mass production unlike software)

■ Iterative view

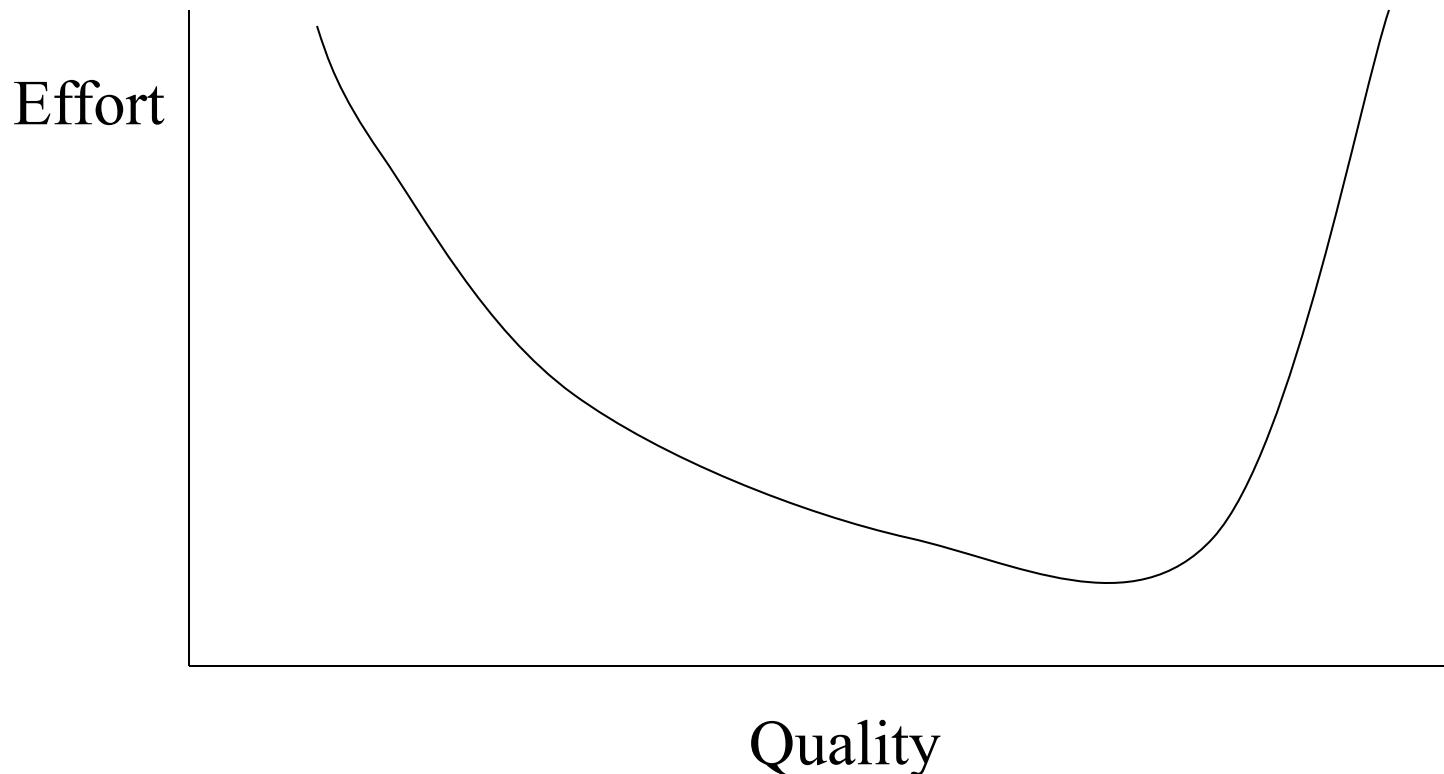
- Feedback and continual improvement is the real heart of quality software

Total quality management

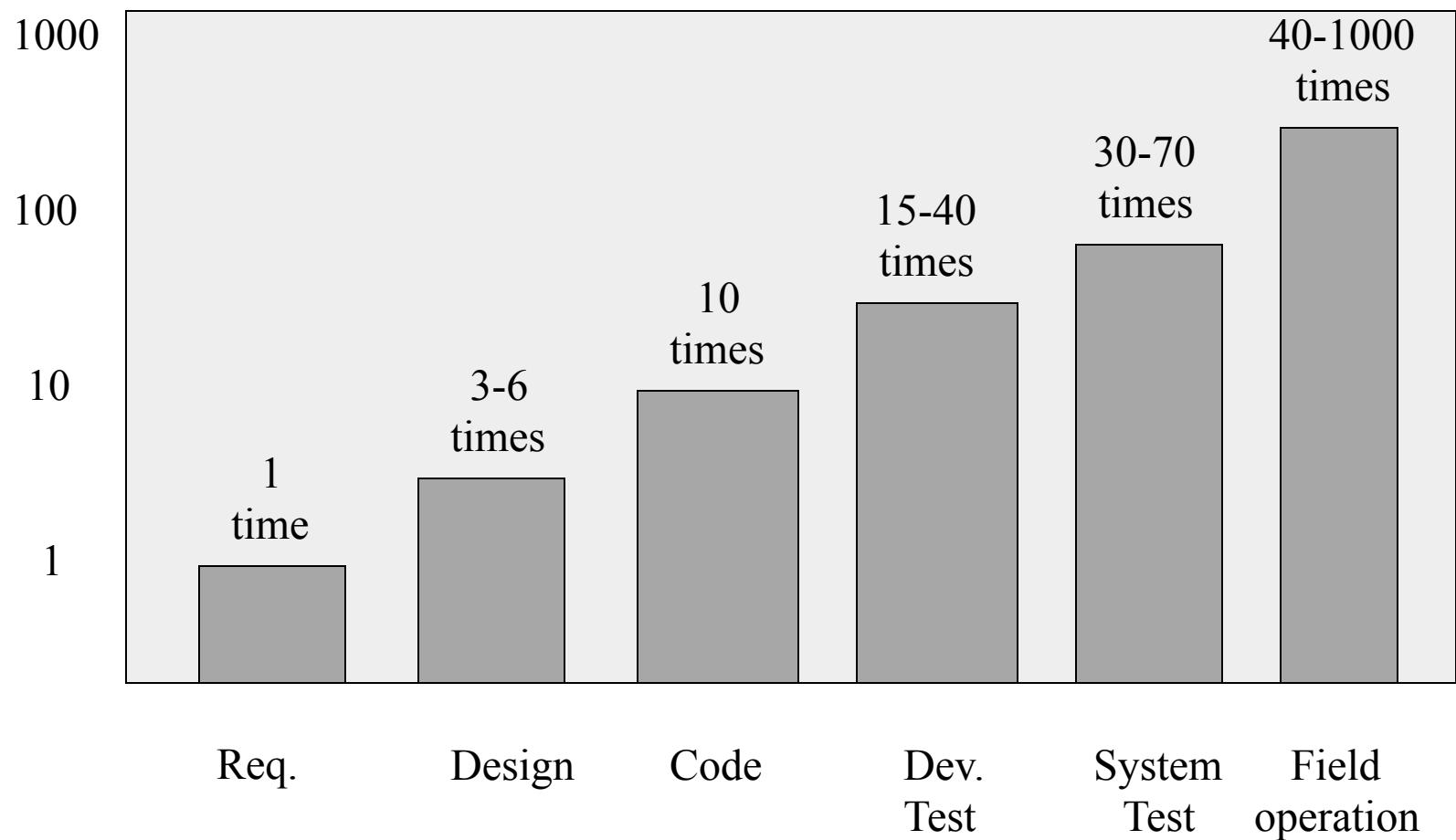


- Factories
- Goal is for every item coming off the assembly line to be perfect
- Management, production, engineering, QA
- Everyone is involved in quality
- Develop a reliable, repeatable process
- Continuously improve the process

“Quality is free”



Cost of fixing an error



Error: Terminology?



- Anomaly
- Bug
- Crash
- Defect
- Error
- Failure/fault/flaw/ “feature”
- ...

Failure vs. flaw



- Failure - program didn't work right
- Flaw - mistake in the text of the program
- Failure analysis (debugging) - what flaw caused this failure?
- Flaw analysis - what is wrong with our process that allowed this flaw to be created and not detected?

Failure costs



- Internal
 - Rework
 - Repair
 - Failure analysis
- External
 - Resolving complaints
 - Returning and replacing product
 - Help line

Prevention costs



- Prevention
 - Planning
 - Managing and collecting information
 - Reviews
- Appraisal
 - Inspection
 - Testing

Johnson's law



“If you don’t test for it, your system
doesn’t have it.”

Is it easy to use?

Easy to maintain?

Does it crash?

Does it match the documentation?

Does it make customers happy?

Ways not to improve quality



- Say “Be more careful!”
- Say “Quality is important.”
- Find out whose fault it is and fire him

How to improve quality



- Measure and compare
- Determine root cause of problems
- Create ways to eliminate problems

Metrics



- If you don't see it, it doesn't exist
- Measure quality over time (metrics)
- Display in a public place

- Make quality goals, then check to see if you meet them

How to appraise quality



- Requirements
 - Reviews by customers
 - Prototyping
- Analysis and design models
 - Formal reviews, inspections
- Current system
 - Bug reports
 - User tests
 - Surveys

Bug tracking



- Keep track of
 - Who reported the bug (the failure)
 - Description of the failure
 - Severity
 - The flaw that caused this failure
 - Who is repairing it
 - The repair

Bug tracking



- Use information about failures to estimate reliability
- Compare
 - Critical nature of failure
 - Iteration failure discovered
 - Module that had the flaw

Use quality information to make decisions



- “Must repair all level 1 failures before shipping”
- “Half of all level 1 and 2 failures in the alpha release were in the Call Processing module; we should rewrite it.”
- “Half of all level 1 and 2 defects found in the design reviews were in Call Processing; we should rewrite it.”

Bug tracking



- Discover the flaw (defect) that caused each bug
- Categorize flaws
- Look at categories with the most flaws and improve your process to eliminate them

Technical reviews



- A way to evaluate the quality of requirements, designs, and software
- A way to improve the quality of requirements, designs, and software
- A way to educate new developers and ensure that developers are consistent
- *Proven to be cost-effective!*

Main goal: Evaluate quality



- Produce a report describing
 - Potential problems
 - Summary of overall quality
 - Pass/fail
- Evaluated by expert outsiders
 - Must know enough
 - Shouldn't know too much

Secondary goal: Improve quality



- Find flaws
- Enforce standards
- Improve standards
- Provide feedback to management

The review team



- Leader (moderator)
- Recorder
- Reviewers

Leader



- Responsible for obtaining a good review - or reporting why a good review wasn't possible
- Good review - one that accurately describes the quality of the product 
- Make sure that reviewers have all the material they need for the review
- Get a time and place for the review

Recorder



- Responsible to provide information for an accurate report of the review
- Typically writes notes on a “flip chart” or other public medium
- At end of review, recorder gives summary and makes sure the team agrees
- Recorder helps leader make final report

Reviewers



- Study product in advance and take notes
- Have a check-list of review criteria
- Give both ³positive and negative comments
- Raise issues, don't resolve them 
- Must be technically competent
- Stick to standards - or stick the standards

Result of review



■ Review summary

- Who, what, when and the conclusion

■ Issues list

- Can result in more detailed reports ✓
- Give priority to issues ✓
- Can be disagreement on issues ↙
- Most issues are about product, but can also be about process or standards

Walkthrough



- Producer guides reviewers through the product
- Easier on reviewers
 - Can cover more material
 - More reviewers can participate
- Good for training the reviewers
- Not as good for evaluating and improving the product

Inspection



- Confine attention to a few aspects of the product, one at a time
- Less preparation by reviewers than for review
- More preparation by leader - must give detailed instructions to reviewers

Reviewing products



- Must compare each product with the products that it is based on
 - Class diagram based on use cases
 - Code in RUP based on design
 - Code in XP based on tests
- Some products not based on any other products
 - User stories?
 - Vision statement?

Homework 4



- Review work of another project
 - Pair with someone on your project
 - Swap reviews with a pair from a different project (TAs assigned, see Wiki)
 - Meet once to review their work
 - Meet once to review your own work

To be reviewed



- Select work to be reviewed
 - Should take two hour meeting to go over it
- Give it to other pair to read in advance
- One of you moderates the meeting, the other records
 - Not how it should be done
 - We do it this way to minimize work
- Produce a report of all the issues that were raised

To review



- Make a check list of things to look for
- Read the work, make comments
- Evaluate according to check list
- Meet and go over items you have found
- How should the check list be improved?

In real life



- Usually there are several meetings until all issues are resolved
- Project has a policy that determines what would be reviewed
- “Passing review” is a measure of progress
- Reviews improve checklists, not just the product under review

Example checklist: Smalltalk code



- Class names make sense
- There is a good class comment that defines variables
- Classes not too big (less than 20 methods)
- Classes have behavior
- Method names
 - Spelled out in full
 - Accessors are “attribute” and “attribute:”
 - Methods with side effects are commands
 - Methods without side effects are adjectives or noun phrases
- Methods not too big (less than 15 lines)

Summary



- Reviews are an important quality control technique
- Pair-programming is a kind of informal review
- Use reviews to improve your process, not just your product

Next: DFD or UI design?



■ Data Flow Diagrams

- Hamlet and Maybee, Chapter 17

■ User Interface Design

- Book by Joel Spolsky, available online
- <http://www.joelonsoftware.com/uibook/chapters/fog000000057.html>

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Topics



- Covered since midterm
 - Specification
 - Design
 - Quality assurance
- Next
 - User interface design
 - Two more processes (besides XP and RUP)
 - Summary

Homework 4



- Review work of another project
 - Pair with someone on your project
 - Swap reviews with a pair from a different project (TAs assigned, see Wiki)
 - Meet once to review their work
 - Meet once to review your own work

To be reviewed



- Select work to be reviewed
 - Should take two hour meeting to go over it
- Give it to other pair to read in advance
- One of you moderates the meeting, the other records
 - Not how it should be done
 - We do it this way to minimize work
- Produce a report of all the issues that were raised

To review



- Make a check list of things to look for
- Read the work, make comments
- Evaluate according to check list
- Meet and go over items you have found
- How should the check list be improved?

In real life



- Usually there are several meetings until all issues are resolved
- Project has a policy that determines what would be reviewed
- “Passing review” is a measure of progress
- Reviews improve checklists, not just the product under review

User interface design



- Important
- Hard
- Isn't covered well by most software development processes

When and who should do?



- After data modeling?
 - Yes, for information systems
 - No, for video games

- By a specialist?
 - Yes, for mass-market software
 - No, for in-house IS systems

Design alternatives



Novice users

- Menus
- Make it look like something else
- Simple

Expert users

- Commands
- Specialize to make users efficient
- Powerful

Design alternatives



- Standard IO vs. new IO
- Existing metaphors vs. new metaphors
- Narrow market vs. broad market

Principle



- UI design is more like film-making than bridge-building
 - About communication
 - Requires understanding audience
 - Requires specialized skills
 - Requires iteration

Book review



- Joel Spolsky on UI Design for Programmers
 - Three chapters per lecture
- Learned Helplessness
 - A user interface is well-designed when the program behaves ...
- Program model vs. User Model
 - User model is simple
- Every time you provide an option, you are asking the user to make a decision

Principle



- UI design is more like film-making than bridge-building
 - About communication
 - Requires understanding audience
 - Requires specialized skills
 - Requires iteration

Golden rules



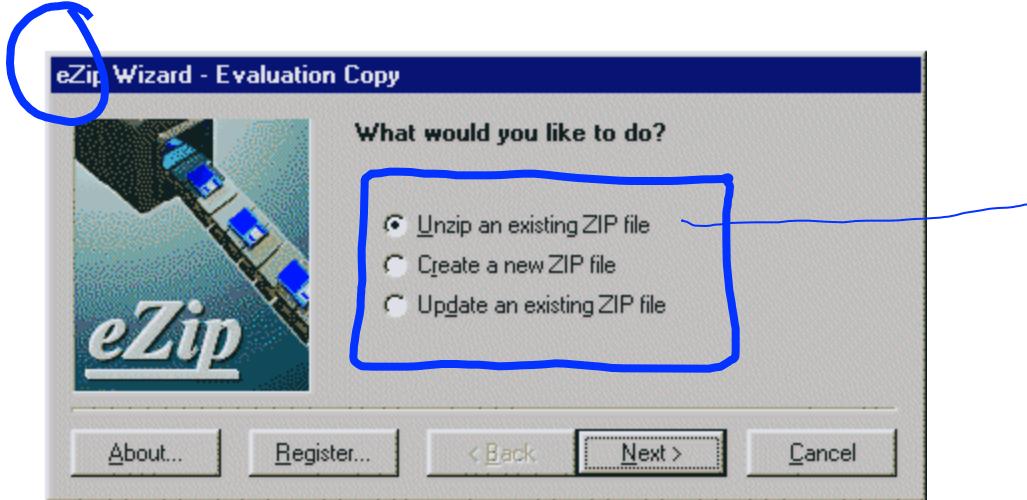
- Place the user in control
- Reduce the user's memory load
- Be consistent

Place the user in control



- No modes (vim?)
 - Use a new window instead of a new mode
 - Make modes visible
- Undo
- Macros
- Hide technical details
- Direct manipulation

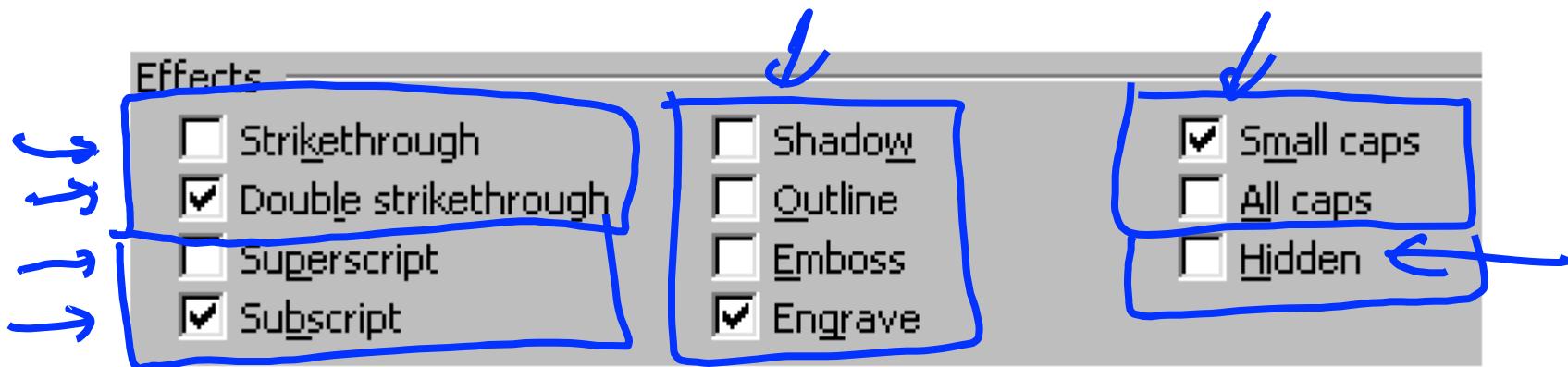
Bad wizards



Obtrusive assistance

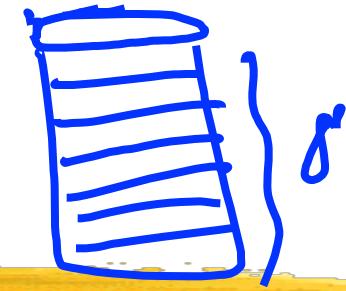


Non-obvious choices

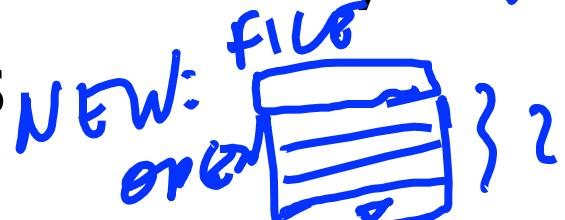


Reduce memory load

old:



- Reduce demand on short-term memory
- Establish meaningful defaults
- Define intuitive shortcuts
- Disclose information progressively
- Use real-world metaphors
- Speak user's language
- Let user recognize, not remember



7±2

Common techniques



- Menus with keyboard shortcuts
- Dialog boxes
- Tabs
- Toolbar

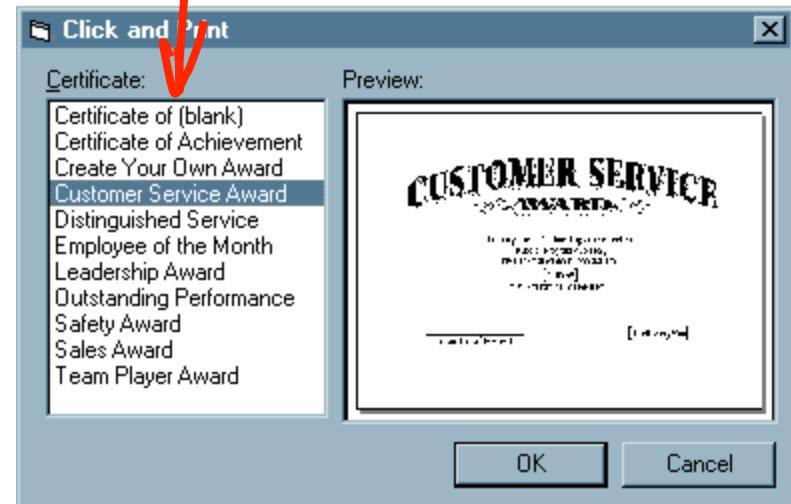
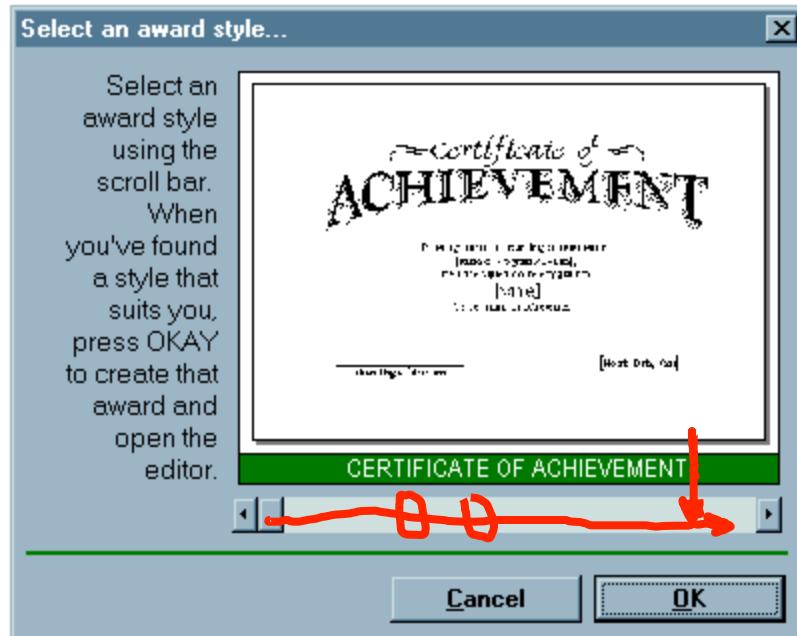
Be consistent



- Use visual interface standards
 - For operating system
 - For organization
 - For product or set of products
- Show context - keep user from getting lost
- System should explain itself

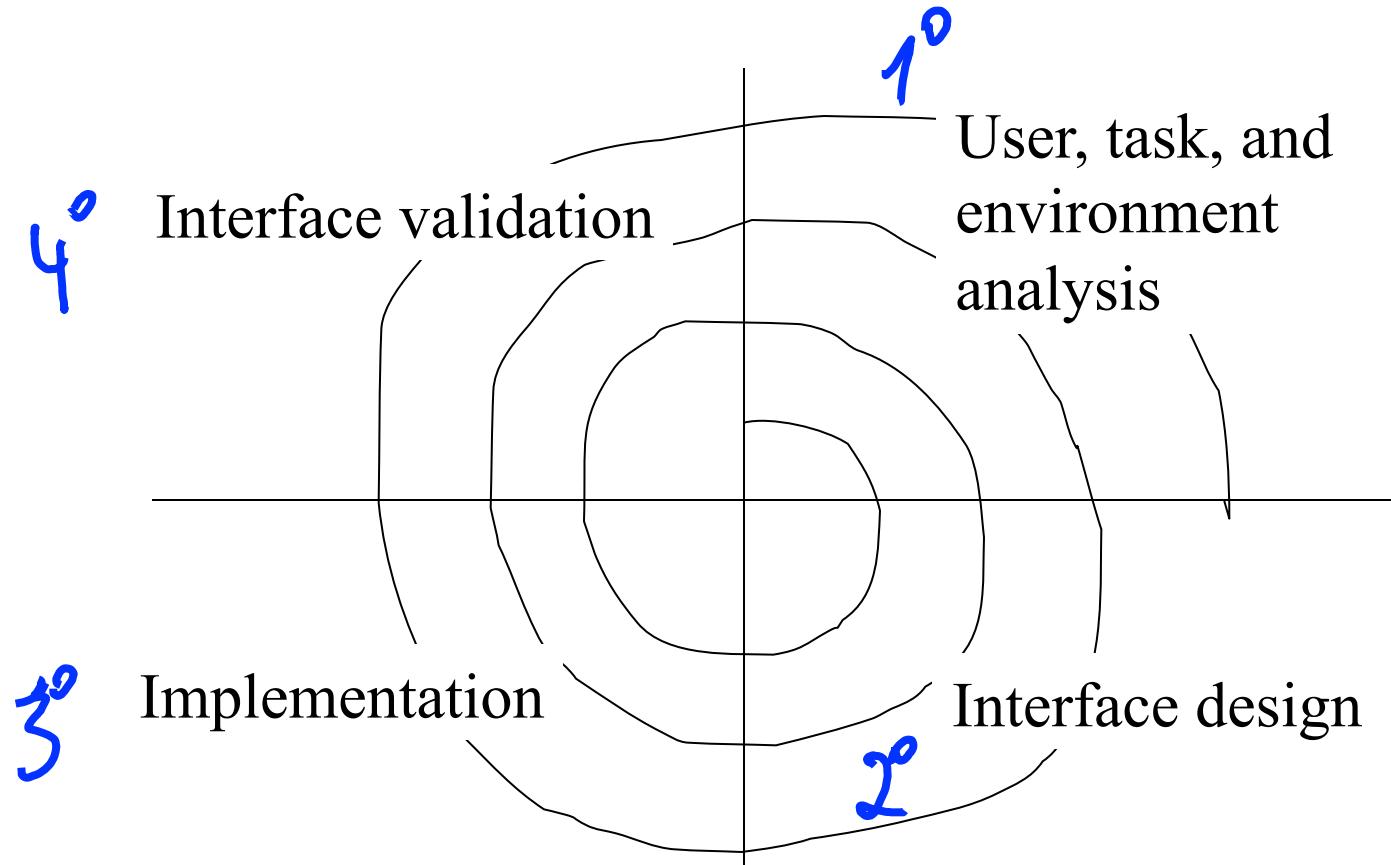


Selection using scroll bar



DON'T KNOW WHAT THEY HAVE
(HOW MANY)
HARD TO NAVIGATE

The UI design process



Models



- Design model - what the designer thinks about the system
- User model - what the user thinks about the system
- System image - interface, manuals, training material, web site

Reconciling models



Pressman says:

“The role of interface designer is to reconcile these differences and derive a consistent representation of the interface”

Early phases



- What are users like?
- What do they think the system should be like?
- What is a single, consistent, model of the system that can satisfy all the users?

Later phases



Design

- | What should system be like?
- | How can we make the users understand it?
- | For each aspect of the system, design the system image to match the desired user model

Validation

- | Does user model match our goal?

Task analysis and modeling



What tasks will a user of the system perform?

High level - why people use the system

Low level - tasks involved in using the system

Tasks and use cases



- Use cases are high-level tasks
 - Decompose high-level ones into low-level ones
 - Find ones that are missing
 - Simplify by generalizing
- UI design requires more detail than use case analysis usually provides

Tasks



- For each task:
 - Is it easy to start the task?
 - Is all the needed information easily accessible?
 - Is it easy to see what to do next?

Low-level design



- Map task into actions that can be directly implemented by standard widgets
- Use consistent labels across tasks
- Use consistent widgets across tasks

User interface design



- UI communicates with the user
- Like any form of communication,
 - Needs feedback and iteration
 - There are standard ways of making a UI
 - Great UIs are rare and require creativity

Next: More UI design



- Read chapters 4-6 of the book on UI design for programmers

<http://www.joelonsoftware.com/uibook/chapters/fog000000060.html>

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative Info



- HW4 postponed, due: Tue, Nov 28
 - You can submit it now if you finished
- HW5 due on Dec 5
 - Document architecture; you can start now
- Final project report due on Dec 7
 - On-campus group can choose to meet with me or the TAs (Jeff for RUP projects, Valas for XP)
- Final exam on Dec 14

Topics on UI design



- Last time
 - Principles
 - Place the user in control
 - Reduce the user's memory load
 - Be consistent
 - Models: design vs. user
- Today
 - Object-oriented UI design
 - Evaluating a UI design
- Next: UI implementation and testing

Joel on software



- Affordance
- Metaphor
- “In most UI decisions, before you design anything from scratch, you absolutely have to look at what other popular programs are doing and emulate that as closely as possible.”

Joel on software



- “Users don't have the manual, and if they did, they wouldn't read it.”
- “In fact, users can't read anything, and if they could, they wouldn't want to.”

UI design



- UI design communicates to user
- User model should match design model
- Bad design model => bad UI design

Single model



- Designers should have a single, coherent model of the system
- Implementation should match that model
- The GUI should communicate that model
- The user model should match the design model

E-mail



■ Tasks

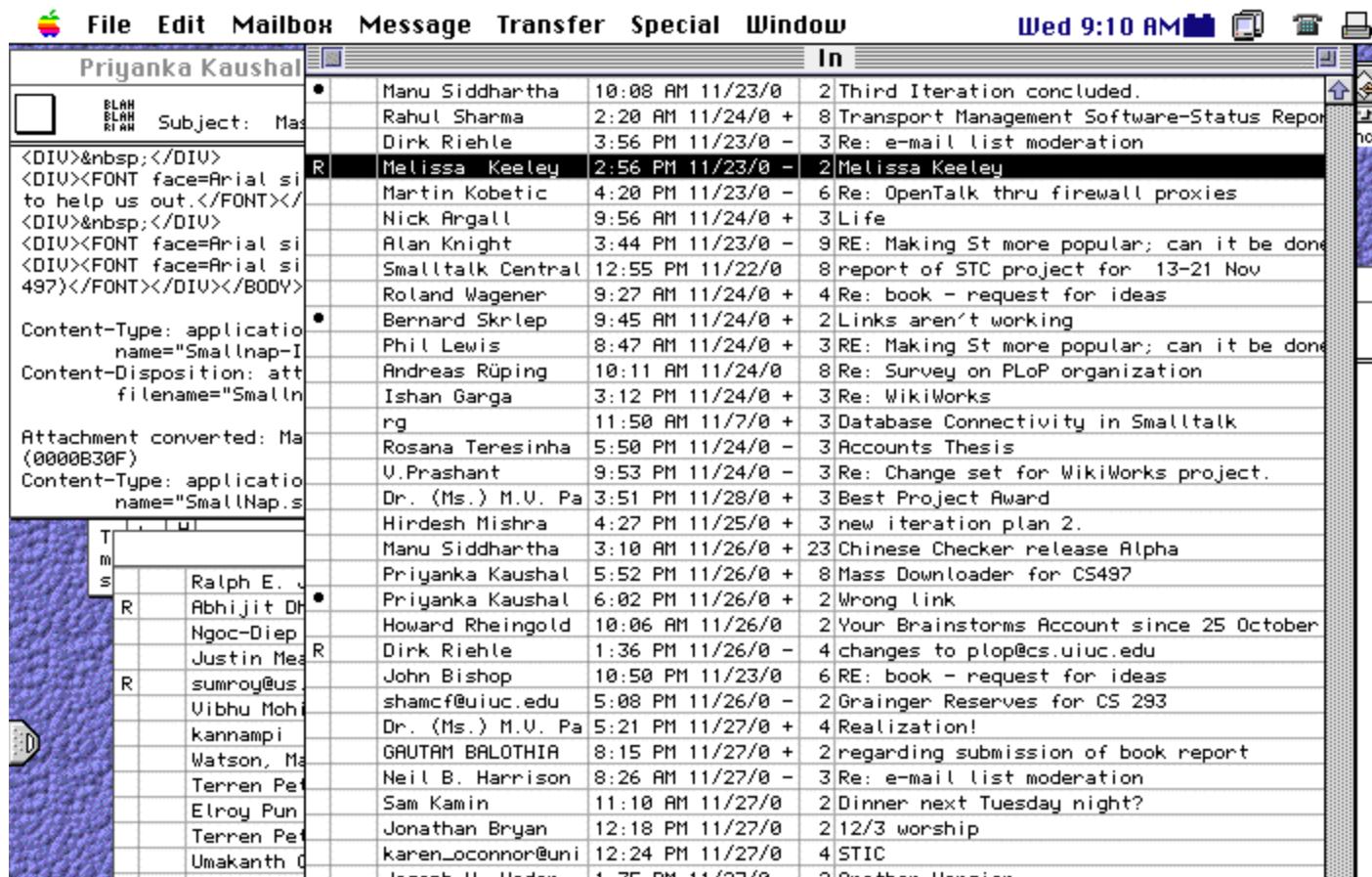
- Read a message
- Check to see if there is more mail
- Reply to a new message
- Send a message to a set of people
- Stop half-way through writing a message and wait till tomorrow
- Save a message

Design model



- Message
- Mailbox
- Incoming messages go to in-box
- Messages in out-box can be marked “ready to be sent”
- New messages created in out-box
- Can move messages from one mailbox to another

Eudora



Object-oriented user interface



- Objects represented by lists, icons
- Operations are on menus, buttons
- Perform operation on selected object
- Make a few operations that work on many kinds of objects
- Specify arguments by:
 - Dialog box
 - Multiple selection

Relate UI to design



- Window for domain object
 - Commands taken from operations on object
 - Direct manipulation interface
- Window for use case
 - Several windows on same domain object
 - Window might display state of use case
 - Window might display argument (dialog box)
 - Commands are steps in use case

Command pattern



- Represent commands by objects
- Command can execute itself
- Executed command knows its result
- Command can undo itself
- System keeps a list of executed commands

Command for “cut text”



- do
 - oldbuffer = editor.getCutbuffer();
 - editor.putCutbuffer(editor.getSelection());
 - editor.putSelection(' ');
- undo
 - editor.putSelection(editor.getCutbuffer());
 - editor.putCutbuffer(oldbuffer);

Command pattern



- Superclass “Command” with interface “do” and “undo”
- Subclass for each kind of command
- Constructors define arguments of the command
- Command subclass defined enough variables to be able to implement undo
- Optional reading: “Design Patterns” by Gamma, Helm, Johnson, and Vlissides

Summary



- Design of a UI for OO systems
- Window for every object
- Window for every use case
- Command pattern

Design choices



- Base GUI on
 - System design (technology)
 - Metaphor
 - Idioms

Optional reading: Alan Cooper

http://www.cooper.com/articles/art_myth_of_metaphor.htm

Base GUI on system design



- Makes GUI design easier
- Makes GUI implementation easier
- Users don't like to see the guts of the system

Base GUI on metaphor



- Can make system easier to learn
- Can limit design
 - Good design has some “magic”
 - May not be a metaphor
 - Bad metaphor worse than none
- XP says (or used to say!) system design should be based on metaphor

Base GUI design on idioms



- People learn new techniques
- If a system has only a few well-designed techniques, it will be easy to learn
- Reuse existing idioms, and design new ones

Response times



From *Usability Engineering* by Jakob Nielsen

- 0.1 sec - limit for “instantaneous”
- 1 sec - limit for “doesn’t interrupt flow”
 - Consider progress indicator
- 10 sec - limit for “keeping attention focused on dialog”
 - Consider making it a background task

Evaluating UI



- Must evaluate UI
 - To see how to improve it
 - To see whether it is good enough to be released

UI metrics



- Size of written specification
- Number of user tasks
- Number of actions per task
- Number of system states
- Number of help messages

UI evaluation



- Once system has users ...
 - Surveys
 - Focus groups
 - Mailing list for support
 - Analyze help desk logs

Early UI evaluation



- Have people use the system
 - Give them tasks
- Find out what is wrong with it
 - Surveys
 - Direct observation
 - Qualitative - did they seem to be having trouble?
 - Quantitative - measure time for tasks

Very early UI evaluation



- Evaluate paper prototypes
- Evaluation team
 - Person to talk to user
 - Person to record observations
 - Person to play computer
- UI made from paper, plastic (pop up menus), and colored ink

UI evaluation



- Be purposeful
 - Decide on purpose of evaluation
 - “Is this menu confusing?”
 - “Can someone start using the system without reading a manual?”
 - Choose tasks
 - Make goals and measure to see if goals are met

Size of evaluation



- Statistically valid sample: 20-100
- Most common size: 5
- Purpose is to invent good UI, not to write a convincing paper
- Perform evaluations after every iteration

Johnson's Law



- “If it hasn’t been tested, it doesn’t work.”
- Applied to UI:
 - If it hasn’t been tested on real users, it is not easy to use.
- Feedback is essential
- Iteration is inevitable

Summary



- User interface design is hard
 - Must understand users
 - Must understand problems
 - Must understand technology
 - Must understand how to evaluate

Next: UI implementation



- Read chapters 7-9 of the book on UI design for programmers

<http://www.joelonsoftware.com/uibook/chapters/fog0000000063.html>

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Homework assignments



- HW4 due today
 - Reviews
- HW5 out, due on Tue, Dec 5
 - Document architecture
- Final project report due on Dec 7 or per agreement with grader
 - On-campus group can choose to meet with me or the TAs (Jeff for RUP projects, Valas for XP)

Final exam



- Dec 14: 7pm-10pm
- Let me know ASAP if you have conflicts
- Exam will include all material from the course with emphasis on after-midterm
- Dec 7 (last lecture): course summary, review session

Topics on UI design



■ Previous

- Principles and rules
- Models: design vs. user
- Object-oriented UI design

■ Today

- Evaluating a UI design
- UI implementation and testing

Evaluating UI



- Must evaluate UI
 - To see how to improve it
 - To see whether it is good enough to be released

UI metrics



- Size of written specification
- Number of user tasks
- Number of actions per task
- Number of system states
- Number of help messages

UI evaluation



- Once system has users ...
 - Surveys
 - Focus groups
 - Mailing list for support
 - Analyze help desk logs

Early UI evaluation



- Have people use the system
 - Give them tasks
- Find out what is wrong with it
 - Surveys
 - Direct observation
 - Qualitative - did they seem to be having trouble?
 - Quantitative - measure time for tasks

Very early UI evaluation



- Evaluate paper prototypes
- Evaluation team
 - Person to talk to user
 - Person to record observations
 - Person to play computer
- UI made from paper, plastic (pop up menus), and colored ink

UI evaluation



- Be purposeful
 - Decide on purpose of evaluation
 - “Is this menu confusing?”
 - “Can someone start using the system without reading a manual?”
 - Choose tasks
 - Make goals and measure to see if goals are met

Size of evaluation



- Statistically valid sample: 20-100
- Most common size: 5
- Purpose is to invent good UI, not to write a convincing paper
- Perform evaluations after every iteration

Johnson's Law



- “If it hasn’t been tested, it doesn’t work.”
- Applied to UI:
 - “If it hasn’t been tested on real users, it is not easy to use.”
- Feedback is essential
- Iteration is inevitable

More from Joel Spolsky



- Design for people who have better things to do with their lives
- Text?
- Mice?
- Memory?

Implementation concerns



- Simplicity
- Safety
- Use standard libraries/toolkits
- Separate UI from application

Simplicity



Don't compromise usability for function

A well-designed interface fades into the background

Basic functions should be obvious

Advanced functions can be hidden

Make controls obvious and intuitive



- Is the trash-can obvious and intuitive?
- Are tabbed dialog boxes obvious and intuitive?
- Is a mouse obvious and intuitive?

Safety



- Make actions predictable and reversible
- Each action does one thing
- Effects are visible
 - User should be able to tell whether operation has been performed
- Undo

Use standard libraries



- Don't build your own!
- If necessary, add to it, but try to use standard parts instead of building your own

Use standard libraries



- Provide familiar controls
- Provide consistency
- Reduce cost of implementation
- Library designers probably better UI designers than you are

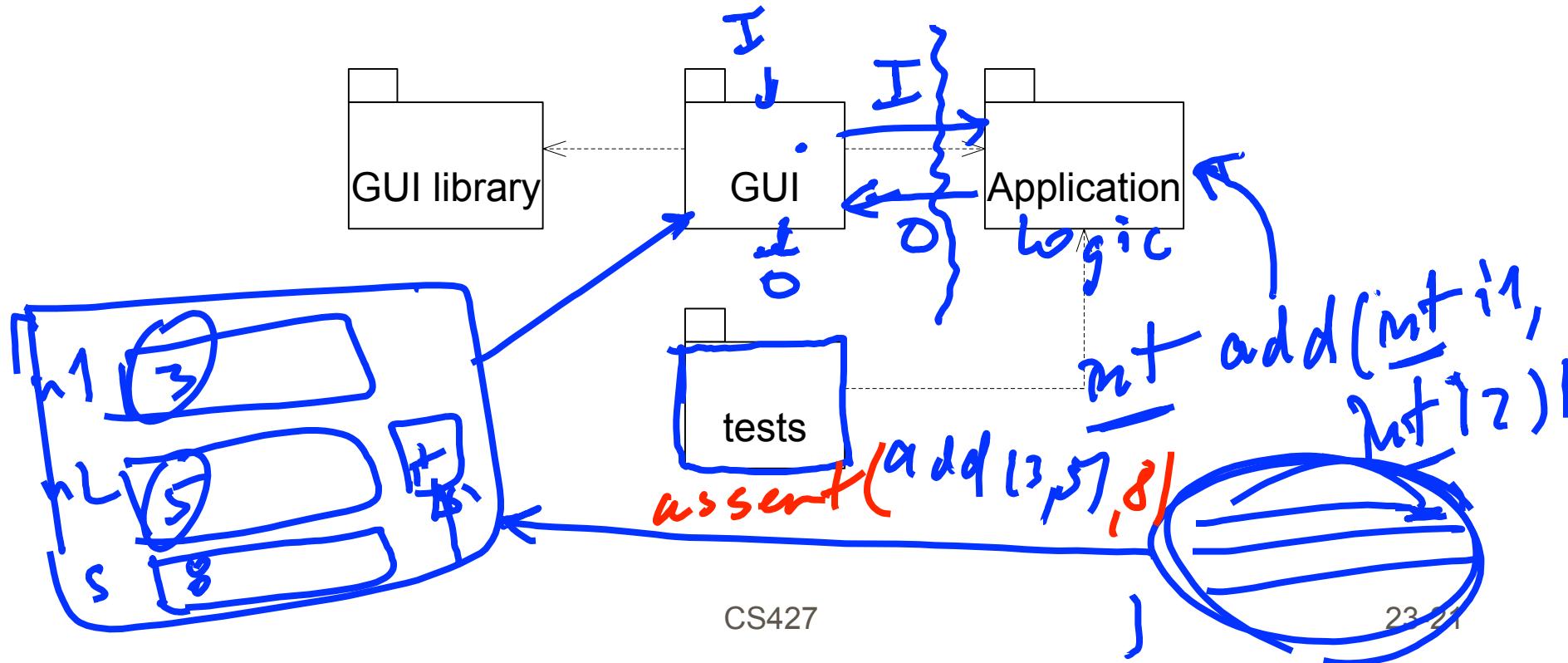
When to build your own



- You are a platform provider or
- You have special needs and a lot of money and
- You are not in a hurry and
- You know what you are doing

Separate UI from application

- UI and application change independently
- UI and application built by different people



UI in ASP



- ASP – embed Visual Basic code in your HTML
- VB can call other code
 - COM objects
 - MTS
- Question – how much VB goes in the web page, and how much goes outside?

Separate UI from application



- HTML is UI
- Put as little VB on web page as possible
- ASP has just enough VB to call a COM object

Benefits



- Write automatic tests for COM object, not for UI
- People who write ASP don't need to know how to program well
- Programmers don't need to be good UI designers

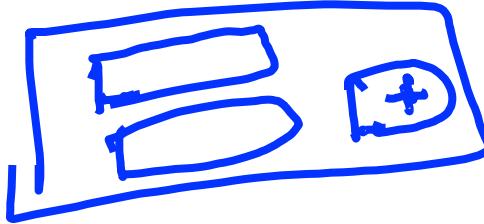
Downside



- COM objects generate HTML
 - But you can make standard set of “adapters” and so don’t have to duplicate code
 - Lists, radio buttons, etc.
- Code tends to creep into ASP
 - Refactor
 - Review

UI in Java

class MyWin
extends JPanel



class App

AWT

SWT

- Learn to use your library (Swing)
- Separate application objects from UI objects
 - UI objects should just display or respond to events
 - Put as much logic as possible in application objects
 - Close-box clicked: UI
 - Has-changed: application object

Results



- Easier to test
 - Automatic tests for application objects
 - Test GUI manually, and write automatic “smoke tests”
- Easier to change
 - Can change “business rule” independently of GUI
 - Can add web interface, speech interface, etc.

Summary of separation



- Separate code into modules
 - To enable people to work independently
 - To make system easier to change
 - To make work easier to reuse
- Special case: Separate UI code from application code

Last words from Joel



- Invent some users
- Figure out the important activities
- Figure out the *user model* -- how the user will expect to accomplish those activities
- Sketch out the first draft of the design
- Iterate over your design again and again, making it easier and easier until it's well within the capabilities of your imaginary users
- Watch real humans trying to use your software. Note the areas where people have trouble, which probably demonstrate areas where the program model isn't matching the user model.

UI Patterns



■ A catalog of techniques

- | Organizing content
- | Getting around
- | Organizing the page
- | Getting input from users
- | Showing complex data
- | Commands and actions
- | Direct manipulation
- | Neither rules nor process
- | Optional reading

<http://time-tripper.com/uipatterns/index.php>

Next: Open-source



- Read
 - Cathedral and the Bazaar
 - <http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/>

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Project grading



- | Final report due on Dec 7
- | Demo scheduled between Dec 4 and 18?
 - | The sooner the better
- | Default: Everyone on team gets the same grade for the project
- | BUT: We will also ask you for more input
 - | Previous semesters: Individual project grades ranged from 0 to over the team grade

Grade curve from 2001



A+	7
A	15
A-	27
B+	19
B	13
B-	11
C+	4
C	1

Topics since midterm



- Specifications
- Design
- Quality assurance
- User interface
- Two more processes
 - Open source (today)
 - Crystal light (next lecture)

Open source



- Reading for this lecture

- The Cathedral and the Bazaar

- <http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/>

- Source code is available

- For inspection

- Independent peer review

- Rapid evolution

Open source



- Technology
 - Internet (communication)
 - Widely used languages
 - Lots of potential users/helpers
 - Standards
- Process

Process



- Cathedral vs. bazaar
- Centralized vs. decentralized
- Planned vs. unplanned

Open source



■ Roles

■ Leader

- | Develops initial system
- | Does what nobody else does
- | Makes final decisions

■ User/programmer

- | Does most of the work

The leader



- An open source project needs a leader
- Delegates as much as he can
- Empowers users
- Decides what goes in

The users



- Users are co-developers
- Users are programmers who can
 - Add features
 - Fix bugs
 - Port to new hardware/operating systems
 - Improve design

Requirements



- Who decides what features get added?
 - Programmers
 - Who want to use the feature (scratch an itch)
 - Who are persuaded to add it
 - Must be a way to distribute changes for a feature
 - Must be way to talk about desired features

Design



- Design is incremental
- Refactoring is important

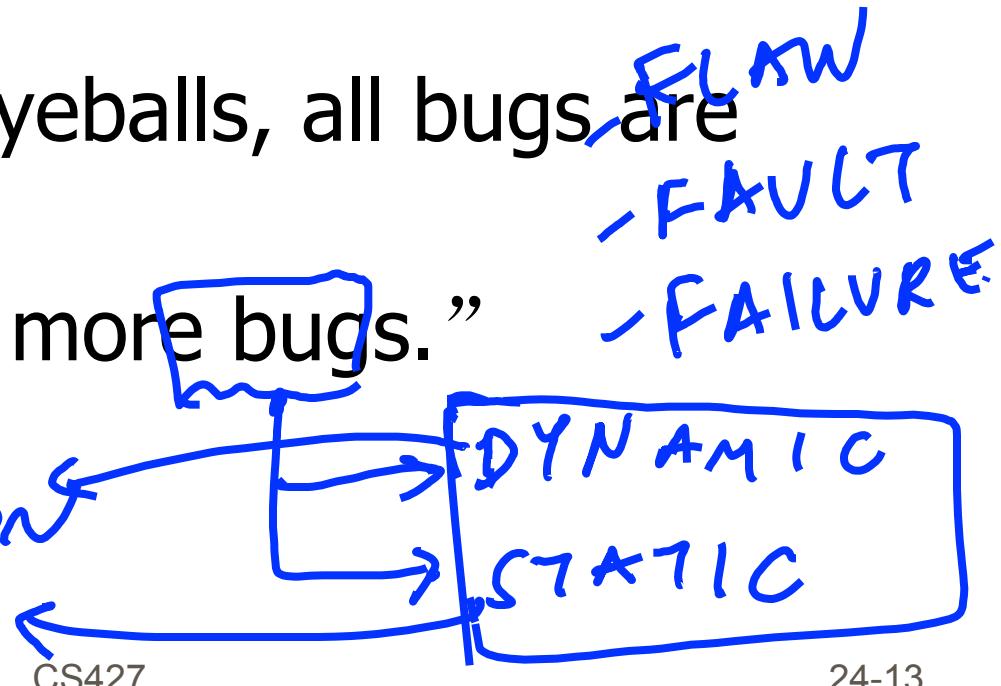
Testing

- Every user is a tester
- Every programmer is a reviewer and bug fixer
- “Given enough eyeballs, all bugs are shallow.”
- “More users find more bugs.”

IN CORRECT EXECUTION
INCORRECT CODE

(MORE END USERS FIND MORE CRASHES)

MORE PROGRAMMERS FIND MORE CODE ERRORS



Life-cycle



- Plausible promise - must start with a (small) working program
- Release early and often
- Recognize good ideas from users
- Keep users connected, let them see the results of their work

Rewards



- Why would anybody do this?
 - They need the program
 - Ego boost
 - Contributing
 - Having people think they are good

Commercial rewards



- Why would anybody do this?
 - Produce better software
 - Produce software more cheaply

Costs



- Need a leader
 - A lot of work over a long time
 - Must communicate
 - An organizer as much as a designer

Enablers



- Internet
 - Copyleft
 - C
 - Boring, high-paying jobs
- 

XP

— INCREMENTAL DESIGN
— REFACTORING

- How is the “Bazaar” process like XP?
- Is the Bazaar leader like an XP customer? ✓
- What about unit tests? ←
- Where does planning happen in the Bazaar? ✓
- What could the Bazaar borrow from XP? ✓
- What could XP borrow from the Bazaar? ✓

Summary



- “Human beings take pleasure in a task when it falls in an optimal-challenge zone; not so easy as to be boring, not too hard to achieve.”
- “A happy programmer is one who is neither underutilized nor weighed down with ill-formulated goals and stressful process friction.”
- “Enjoyment predicts efficiency.”

Next time



- Crystal Clear by Alistair Cockburn
 - Reading will be updated on Wiki and sent to newsgroup

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- Wiki down again 😞
 - If you were affected, you can submit HW5 on Thursday, Dec 7
- Schedule demo between today and 18?
 - The sooner the better
 - Final report due on Dec 7 or by arrangement
 - All team members should be present or be able to explain work of the missing members
 - Default: Everyone on team gets the same grade for the project, BUT there can be variance

Final exam



- Thu, Dec 14, 7-10pm
 - 1310 DCL
 - 269 Everitt
- Conflict exam: Mon, Dec 11, 1:30-4:30pm
 - 3403 SC
- More info will be on Wiki

Topics since midterm



- Specifications
- Design
- Quality assurance
- User interface
- Two more processes
 - Open source (previous lecture)
 - Crystal clear (today)

Crystal clear



- Reading for this lecture
 - Crystal light methods overview
http://alistair.cockburn.us/index.php/Crystal_light_methods
- A family of processes
 - Named after crystal terminology
 - Color and hardness
 - Crystal Clear: The Simplest Process That Can Work (Applied anthropology)

Crystal clear



- “Good, small teams have been putting out good software for decades, using thinking, communicating and delivering as their primary tools.”
- “Good, small teams do not scale up in number, so keep them small, permit them to move fast, do project tracking through their final results, not their intermediate thinking.”

Crystal clear assumptions



- People are good at looking around
- People are not as tidy as abstractions of them make them appear
- Communication involves a lot more than the words spoken or written
- Software construction is growing understanding of problem and solution

Crystal Clear assumptions



- Actual, working processes are
 - Extremely complicated
 - Hard to write down
 - Hard to follow
 - Likely to be wrong when written down
- Methodologists are prone to overcomplicate or embellish things

Crystal Clear



- Make process as simple as you can
- Rely on communication, individual skill
- Size: 4-6
- Essential moneys, but not life

People



- Sponsor
- Senior designer
- User
- Designers

Other roles



- Can be separate people, or can be one of the designers, perhaps the senior designer
- Business expert
- Coordinator
- Tester
- Writer

Policies



- Use increments for project staging, tracking by milestones and predicted risks
- Involve user directly
- Requirements are annotated usage scenarios
- Peer code reviews

Policies



- Code ownership model
- Regression testing framework
- Code standard
- User interface standard

Work products



- Not a substitute for understanding
 - Understanding is primary
 - Work products are secondary
- A substitute for discipline
- A “minimal set”

Work products



- Methodology
- Team structure
- Release sequence
- Viewing and release schedule
- Risk list
- Project status

Requirements work products



- Mission statement
- Actor-goal pairs
- Annotated use cases
- Requirements file

Design work products



- System design
- Common object model
- Screen drafts
- Design sketches
- Source code
- Migration code

Tests and final system



- Test cases
- Test results
- Packaged system
- User manual

Making it work



- How do you make sure people communicate?
- How do you decide on an ownership model, coding standard, etc?

Requirements



- Where do requirements come from?
- How are they recorded?
- How are they validated?
- How are they used?

Common object model



- How is the object model recorded?
- How is the object model created?
- How is the object model used?

Summary



- Is XP just a special case of Crystal Light?
- Could a RUP project also be a Crystal Light project?
- Could a Bazaar project be a special case of Crystal Light?

Next: Course summary



- Last lecture on Thursday, Dec 7

CS427: **Software Engineering I**



Darko Marinov

(slides from Ralph Johnson)

Administrative info



- All HWs should be turned in

- Final demo schedule

<http://pixie.cs.uiuc.edu:8080/SEcourse/demo+2006>

- Agenda

- Your overall presentation

- Your individual presentations

- Our questions

Common project problems



- Takes too long to learn technology
- Don't know how to divide up work
- A few people do all the work
 - Spend more effort on teaching, less on building
 - Pair for learning
 - All work in the same room at the same time
 - Fire those who are holding you back

Final exam: place and time



- Thu, Dec 14, 7-10pm
 - 1310 DCL (netids starting with a-md)
 - 269 Everitt (netids starting with me-z)
- Conflict exam: Mon, Dec 11, 1:30-4:30pm
 - 3403 SC (by request only!)
- More info on Wiki
<http://pixie.cs.uiuc.edu:8080/SEcourse/Fall+2006+Final+Assignments>

Final exam: topics



- Study guide and finals from 2004 and 2005 on Wiki
<http://pixie.cs.uiuc.edu:8080/SEcourse/Exams+Fall+06>
- Material from first half, but not a lot
- Specifications, design (high and low), and analysis
 - State machines, ADTs, other notations (no DFD!)
 - Modularity, abstraction, information hiding, refinement
- Quality assurance
- User interface design (no UI patterns!)
- RUP and XP
- Open Source and Crystal Clear

Reading: books + papers



- Chapter listing on Wiki
<http://pixie.cs.uiuc.edu:8080/SEcourse/Exams+Fall+06>
- Parnas: modularity
- Spolsky: Joel on UI Design
- Raymond: The Cathedral and the Bazaar
- Cockburn: Crystal Clear
- Don't need to read optional material!
 - But need to know topics from the slides

Grade curve from 2001



A+	7
A	15
A-	27
B+	19
B	13
B-	11
C+	4
C	1

Software engineering I and II



- 427 – process
 - Modeling, management, requirements and design
- 428/9 – tools
 - Software configuration management
 - Testing
 - Metrics
 - Maintenance and reverse engineering
 - Client-server, web systems, component software
 - CASE tools

RUP and XP



- What are the main differences?
- What different assumptions do they make?
- How are their goals different?

Scheduling



- How are the goals of scheduling different in RUP and XP?
- How do the techniques differ?
- Who makes schedules? Does it matter?

Requirements



- How do use cases differ from user stories?
- Who does the work? Does it matter?

Analysis



- What happens during the RUP workflow called “analysis” ?
- XP doesn’t have anybody called an “analyst” or a phase called “analysis”
 - Where does the work get done?
 - What are the advantages and disadvantages of each approach?

Architecture



- What is “Architecture” in RUP?
- What does an “architect” do?
- XP doesn’t have an architect and never mentions architecture
 - Does this work get done anyway?

Design



- What happens during the RUP workflow called “design” ?
- XP doesn’t have anybody called a “designer” or a phase called “design”
 - Where does the work get done?
 - What are the advantages and disadvantages of each approach?

Evaluation



- How do people on an RUP project know whether they are building the right system?
- How do people on an XP project know whether they are building the right system?

Quality



- Do RUP and XP have different ideas of software quality? If so, how?

Manager



- What does an RUP manager do?
- What does an XP manager do?
- Which job is harder?

Developer



- What things does a RUP developer need to know that an XP developer doesn't?
- What things does an XP developer need to know that an RUP developer doesn't?
- Which job is harder?

Other topics



- Specification, design
 - Know things from HW3
- Quality assurance
 - Dealing with bugs
- UI design
 - Good and bad examples
- Open source and crystal clear
 - Comparison with XP and RUP

Your questions?



- You can ask now
- Send emails to marinov or ta427
- Post to the newsgroup
- Office hours on Monday?
10:

CONF 107 EXAM:
1:30