College of Information Studies

University of Maryland Hornbake Library Building College Park, MD 20742-4345

Software Engineering

Week 12

INFM 603

The System Life Cycle

- Systems analysis
  - How do we know what kind of system to build?
- User-centered design
  - How do we discern and satisfy user needs?
- Implementation
  - How do we build it?
- Management
  - How do we use it?

Software Engineering

- Systematic
  - Repeatable
- Disciplined
  - Transferable
- Quantifiable
  - Managable

Prehistoric Software Development

- Heroic age of software development:
  - Small teams of programming demigods wrestle with many-limbed chaos to bring project to success … sooner or later … maybe …
  - Kind of fun for programmers ...
  - … not so fun for project stakeholders!

The Waterfall Model

- Key insight: invest in the design stage
  - An hour of design can save a week of debugging!
- Three key documents
  - Requirements
    - Specifies what the software is supposed to do
  - Specification
    - Specifies the design of the software
  - Test plan
    - Specifies how you will know that it did it

The Waterfall Model

Coding

- Coding standards
  - Layout (readable code is easier to debug)
  - Design Patterns
    - Avoid common pitfalls, build code in expected manner
  - Verification: code checkers
- Code review
  - Computers don't criticize; other coders do!
  - Formalized in pair programming
  - (Proofs of correctness)
- Code less
  - Bugs per 100 lines is surprisingly invariant
  - Libraries: maximise re-use of code, yours and others

Coding Standards Examples

- Use set and get methods
  - Limits unexpected “side effects”
- Check entry conditions in each method
  - Flags things as soon as they go wrong
- Write modular code
  - Lots of method calls means lots of checks

Version Control

- Supports asynchronous revision
  - Checkout/Checkin model
  - Good at detecting incompatible edits
  - Not able to detect incompatible code
- Revision Tree
  - Named versions
  - Described versions
- Standard tools are available
  - SVN (centralized), git (distributed)
  - Key idea: store only the changes

Types of “Testing”

- Design walkthrough
  - Does the design meet the requirements
- Code walkthrough
  - Does the code implement the requirements?
- Functional testing
  - Does the code do what you intended?
- Usability testing
  - Does it do what the user needs done?

Functional Testing

- Unit testing
  - Components separately
- Integration testing
  - Subsystems
- System testing
  - Complete system (with some coverage measure)

- Regression testing
  - Invariant output from invariant input

Planning Functional Testing

- You can’t test every possibility
  - So you need a strategy
- Several approaches
  - Object-level vs. system-level
  - Black box vs. white box
  - Ad-hoc vs. systematic
  - Broad vs. deep
- Choose a mix that produces high confidence

Planning Usability Testing

- Define one or more scenarios
  - Based on the requirements (not your design!)
  - Focus only on implemented functions
- Provide enough training to get started
  - Usually with a little supervised practice
- Banish pride of authorship
  - Best to put programmers behind one-way glass!
- Record what you see
  - Notes, audiotape, videotape, key capture

Types of Errors

- Syntax errors
  - Detected at compile time
- Run time exceptions
  - Cause system-detected failures at run time
- Logic errors
  - Cause unanticipated behavior (detected by you!)
- Design errors
  - Fail to meet the need (detected by stakeholders)

Bug Tracking

- Bugs are alleged errors
  - System-level or component level
  - Development or deployment
  - True bugs or misuse/misunderstanding
- Bug tracking is needed
  - Particularly on large projects
- Standard tools are available
  - e.g., Bugzilla

Debugging is harder than coding!

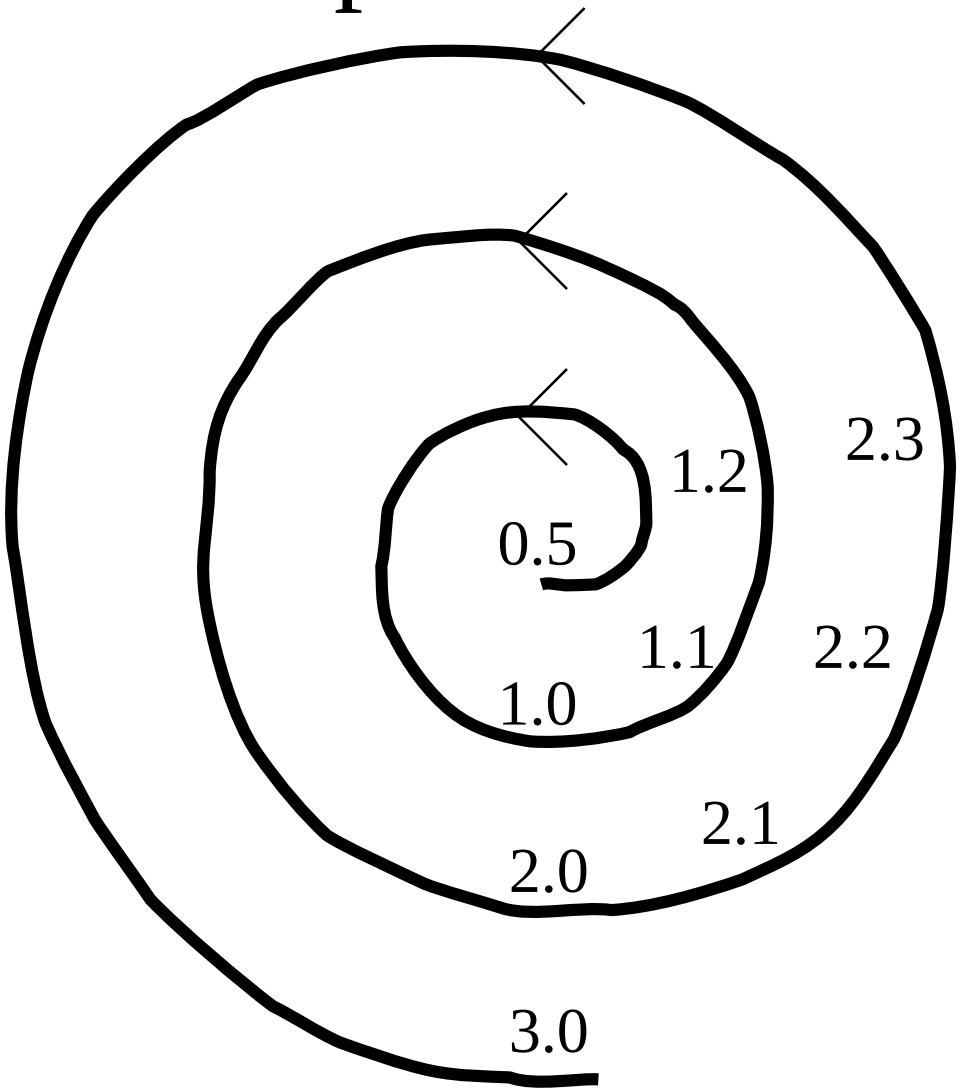
"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it"

– Brian W. Kernighan and P. J. Plauger, *The Elements of Programming*

The Spiral Model

- Build what you think you need
  - Perhaps using the waterfall model
- Get a few users to help you debug it
  - First an “alpha” release, then a “beta” release
- Release it as a product (version 1.0)
  - Make small changes as needed (1.1, 1.2, ….)
- Save big changes for a major new release
  - Often based on a total redesign (2.0, 3.0, …)

The Spiral Model



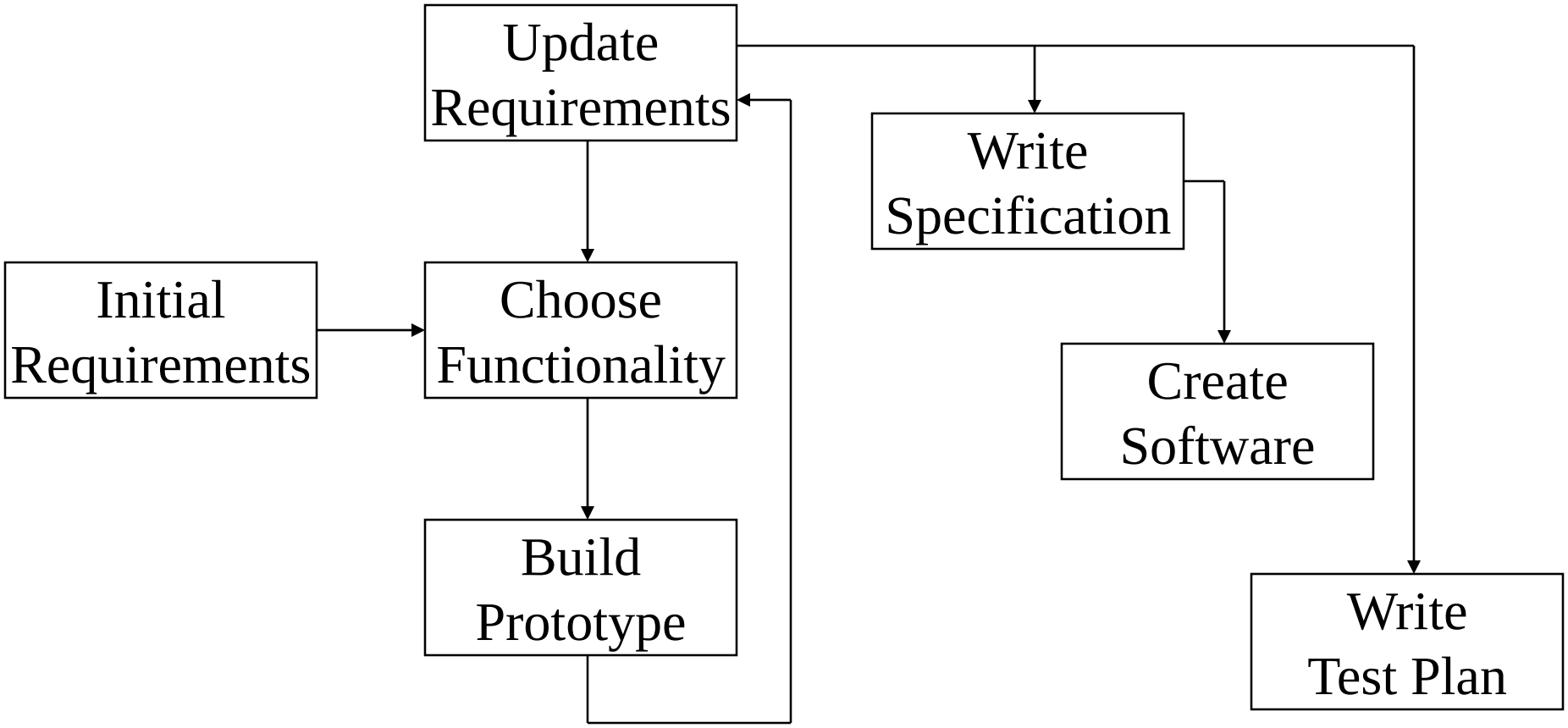
Unpleasant Realities

- The waterfall model doesn’t work well
  - Requirements usually incomplete or incorrect
- The spiral model is expensive
  - Rule of thumb: 3 iterations to get it right
  - Redesign leads to recoding and retesting

The Rapid Prototyping Model

- Goal: explore requirements
  - Without building the complete product
- Start with part of the functionality
  - That will (hopefully) yield significant insight
- Build a prototype
  - Focus on core functionality, not in efficiency
- Use the prototype to refine the requirements
- Repeat the process, expanding functionality

Rapid Prototyping + Waterfall



Objectives of Rapid Prototyping

- Quality
  - Build systems that satisfy the real requirements by focusing on requirements discovery
- Affordability
  - Minimize development costs by building the right thing the first time
- Schedule
  - Minimize schedule risk by reducing the chance of requirements discovery during coding

Characteristics of Good Prototypes

- Easily built (about a week's work)
  - Requires powerful prototyping tools
  - Intentionally incomplete
- Insightful
  - Basis for gaining experience
  - Well-chosen focus (DON'T built it all at once!)
- Easily modified
  - Facilitates incremental exploration

Prototype Demonstration

- Choose a scenario based on the task
- Develop a one-hour script
  - Focus on newly implemented requirements
- See if it behaves as desired
  - The user's view of correctness
- Solicit suggestions for additional capabilities
  - And capabilities that should be removed

A Disciplined Process

- Agree on a project plan
  - To establish shared expectations
- Start with a requirements document
  - That specifies only bedrock requirements
- Build a prototype and try it out
  - Informal, focused on users -- not developers
- Document the new requirements
- Repeat, expanding functionality in small steps

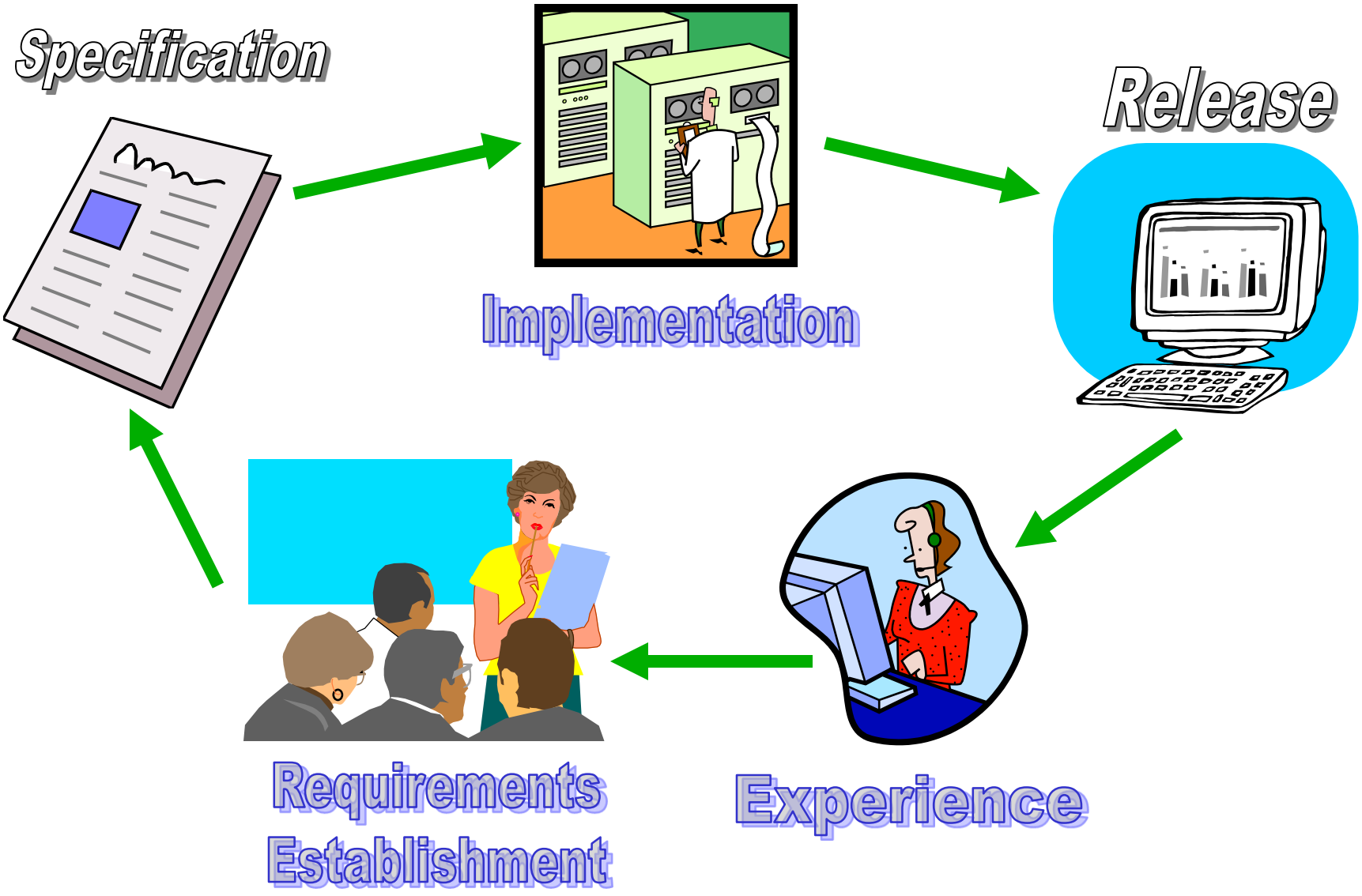
What is NOT Rapid Prototyping?

- Focusing only on appearance
  - Behavior is a key aspect of requirements
- Just building capabilities one at a time
  - User involvement is the reason for prototyping
- Building a bulletproof prototype
  - Which may do the wrong thing very well
- Discovering requirements you can’t directly use
  - More efficient to align prototyping with coding

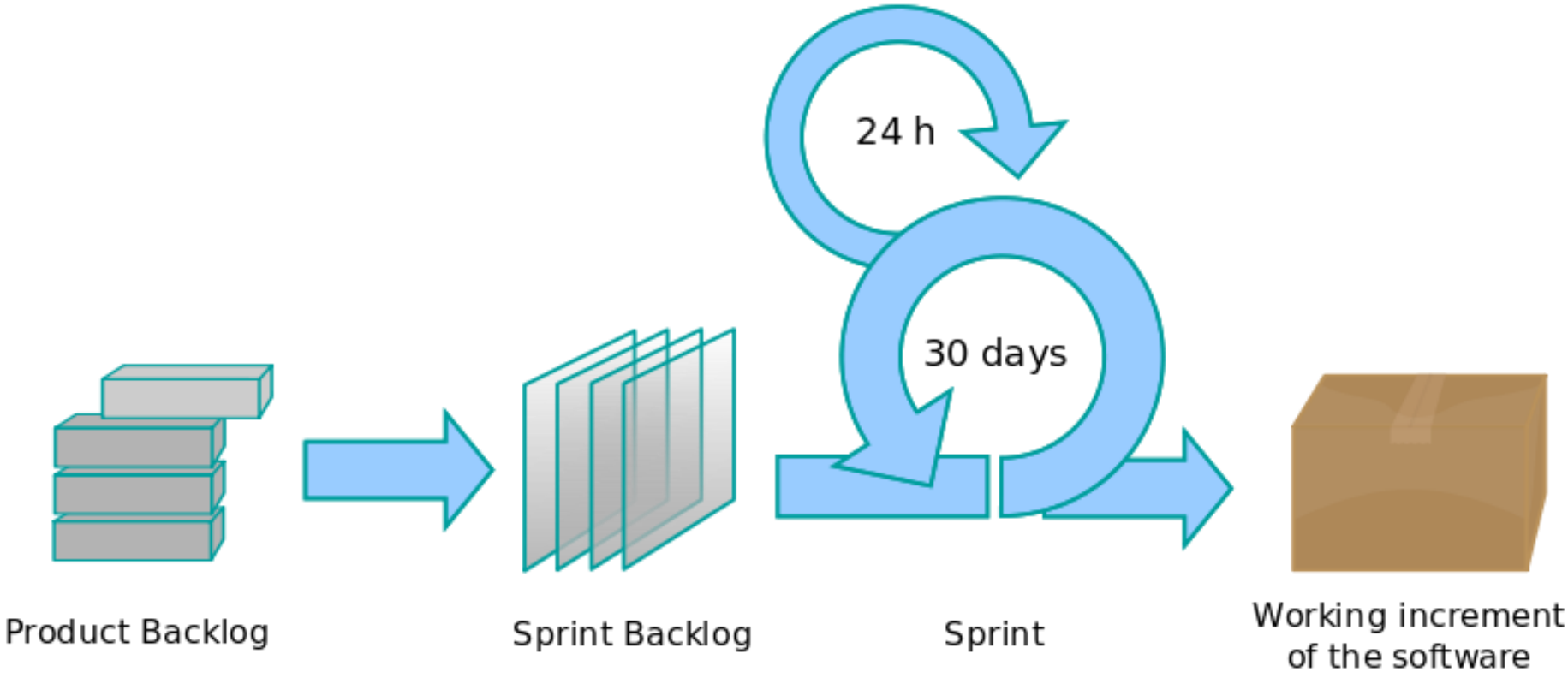
Agile Methods

- Prototypes that are “built to last”
- Planned incremental development
  - For functionality, not just requirements elictitation
- Privileges time and cost
  - Functionality becomes the variable

Agile Methods



SCRUM


24 h
30 days
Product Backlog
Sprint Backlog
Sprint
Working increment
of the software

Basic SCRUM Cycle

- The sprint:
  - Basic unit of development
  - Fixed duration (typically one month)
  - End target is a working system (*not* a prototype)
- Sprint planning meeting
  - Discussion between product owner and development team on what can be accomplished in the sprint
  - Sprint goals are owned by the development team

Disadvantages

- Can be chaotic
- Dependent on a good SCRUM master to reconcile priorities
- Requires dedication of team members
- Slicing by “user stories” isn’t always feasible

SCRUM: Key Concepts

- Roles:
  - Product owner: voice of the customer
  - Development team: small team software engineers
  - Scrum master: primary role as facilitator
- User stories: short *non-technical* description of desired user functionality
  - “As a user, I want to be able to search for customers by their first and last names”
  - “As a site administrator, I should be able to subscribe multiple people to the mailing list at once”

Standup Meetings

- Short, periodic status meetings (often daily)
- Three questions:
  - What have you been working on (since the last standup)?
  - What are you planning to work on next?
  - Any blockers?

Software Quality Assurance Models

- Patterned on other quality assurance standards
  - e.g., ISO 9000
- Focus is on measuring quality of process management
  - Models don't tell you how to write good software
  - They don't tell you what process to use
  - They assess whether **you** can measure your process
    - If you can’t measure it, you can’t improve it!

ISO 15504

ISO 15504 has six capability levels for each process:

1. Not performed
2. Performed informally
3. Planned and tracked
4. Well-defined
5. Quantitatively controlled
6. Continuously improved

Total Cost of Ownership

- Planning
- Installation
  - Facilities, hardware, software, integration, migration, disruption
- Training
  - System staff, operations staff, end users
- Operations
  - System staff, support contracts, outages, recovery, …

Management Issues

- Policy
  - Privacy, access control, appropriate use, …
- Training
  - System staff, organization staff, “end users”
- Operations
  - Fault detection and response
  - Backup and disaster recovery
  - Audit
  - Cost control (system staff, periodic upgrades, …)
- Planning
  - Capacity assessment, predictive reliability, …

Strategic Choices

- Acquisition
  - Proprietary (“COTS”)
  - Open source
- Implementation
  - Integrate “Best-of-breed” systems
  - “One-off” custom solution

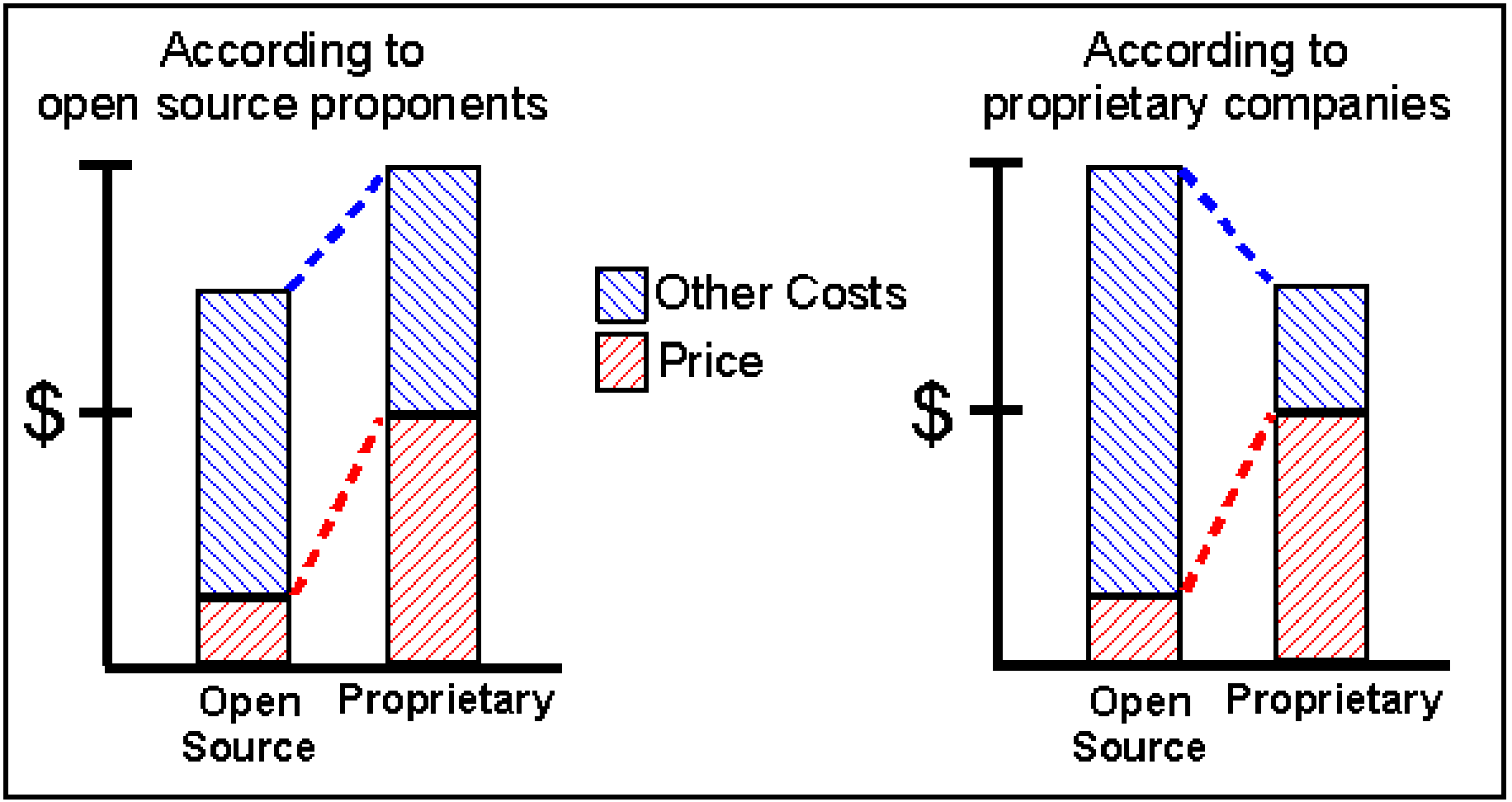
Open Source “Pros”

- More eyes ⇒ fewer bugs
- Iterative releases ⇒ rapid bug fixes
- Rich community ⇒ more ideas
  - Coders, testers, debuggers, users
- Distributed by developers ⇒ truth in advertising
- Open data formats ⇒ Easier integration
- Standardized licenses

Open Source “Cons”

- Communities require incentives
  - Much open source development is underwritten
- Developers are calling the shots
  - Can result in feature explosion
- Proliferation of “orphans”
- Diffused accountability
  - Who would you sue?
- Fragmentation
  - “Forking” may lead to competing versions
- Little control over schedule

Total Cost of Ownership



Open Source Business Models

- Support Sellers

  Sell distribution, branding, and after-sale services.

- Loss Leader

  Give away the software to make a market for proprietary software.

- Widget Frosting

  If you're in the hardware business, giving away software doesn't hurt.

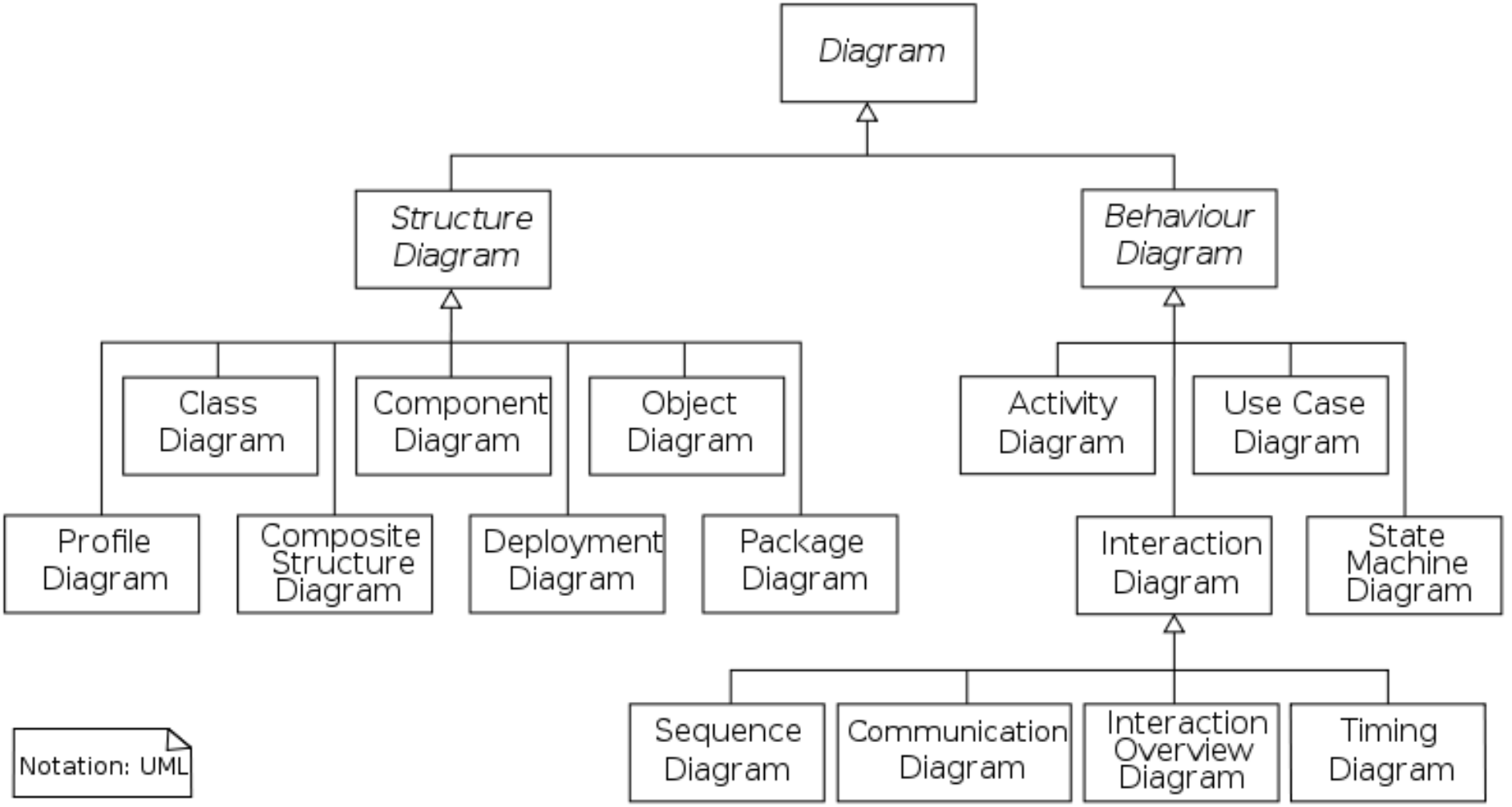
- Accessorizing

  Sell accessories:
  books, compatible hardware, complete systems with pre-installed software

Unified Modeling Language

- Real systems are more complex than anyone can comprehend
- Key idea: Progressive refinement
  - Carve the problem into pieces
  - Carve each piece into smaller pieces
  - When the pieces are small enough, code them
- UML provides a formalism for doing this
  - But it does not provide the process

Unified Modeling Language



Specifying Structure

- Capturing the big picture
  - Use case diagram (interactions with the world)
  - Narrative
  - Scenarios (examples to provoke thinking)
- Designing the object structure
  - Class diagram (“entity-relationship” diagram)
  - Object diagram (used to show examples)

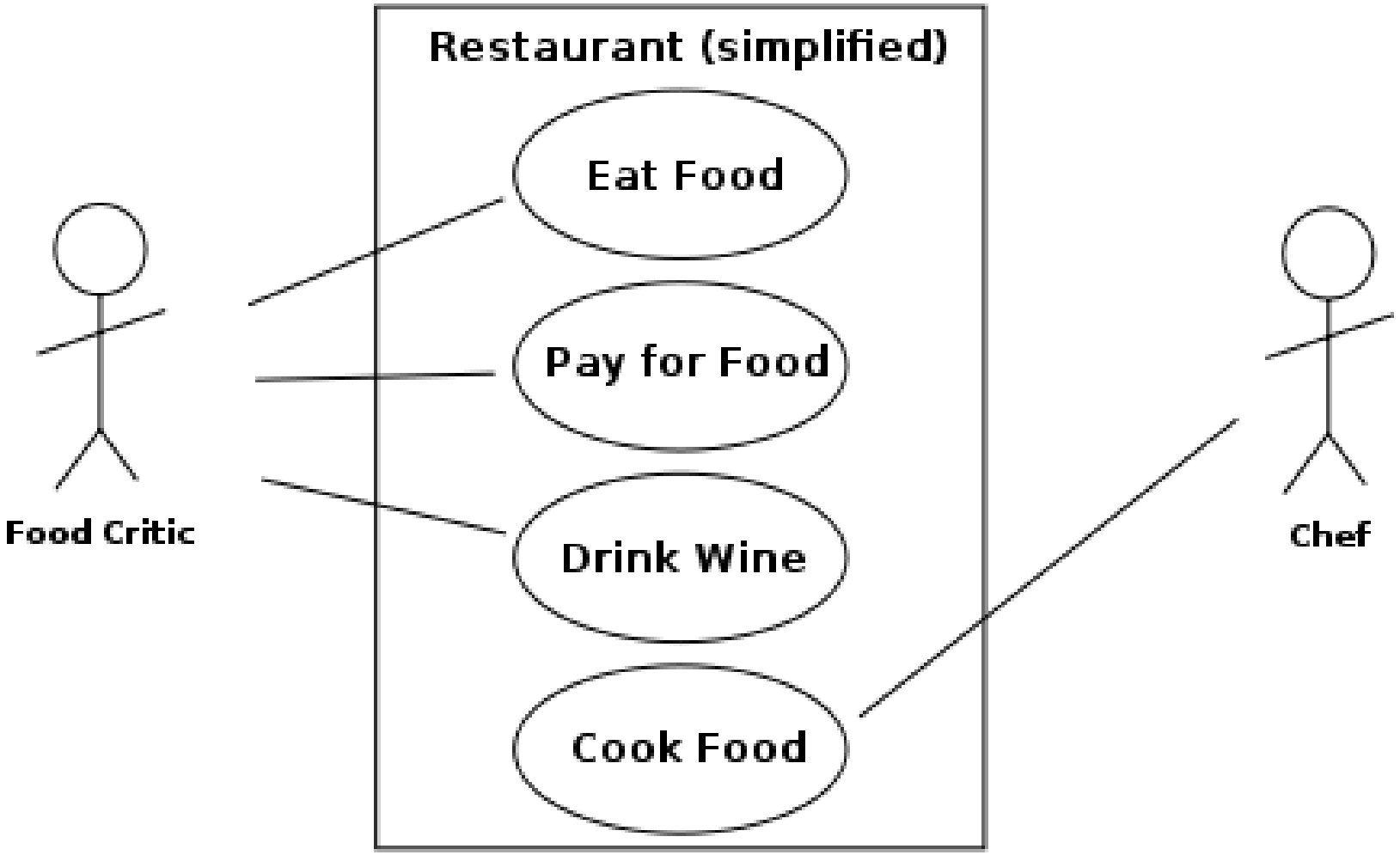
Specifying Behavior

- Represent a candidate workflow
  - Activity diagram (a “flowchart”)
- Represent object interactions for a scenario
  - Collaboration diagram (object-based depiction)
  - Sequence diagram (time-based depiction)
- Represent event-object interactions
  - Statechart diagram (a “finite state machine”)

Use Case Design

- Use Case Diagram
  - Input-output behavior
- Use Case Narrative
  - Explains each use case
- Use Case Scenario
  - Activity diagram shows how the use cases are used together

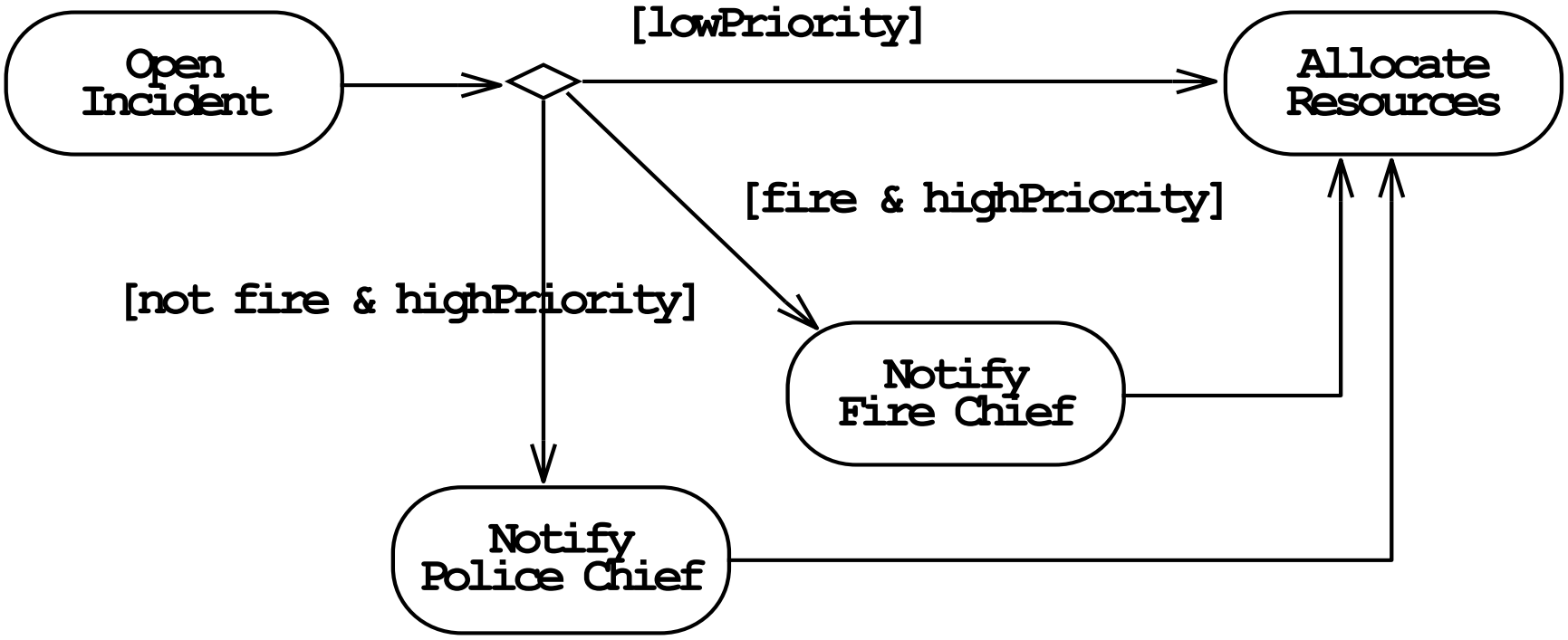
Use Case Diagram



Use Case Diagram

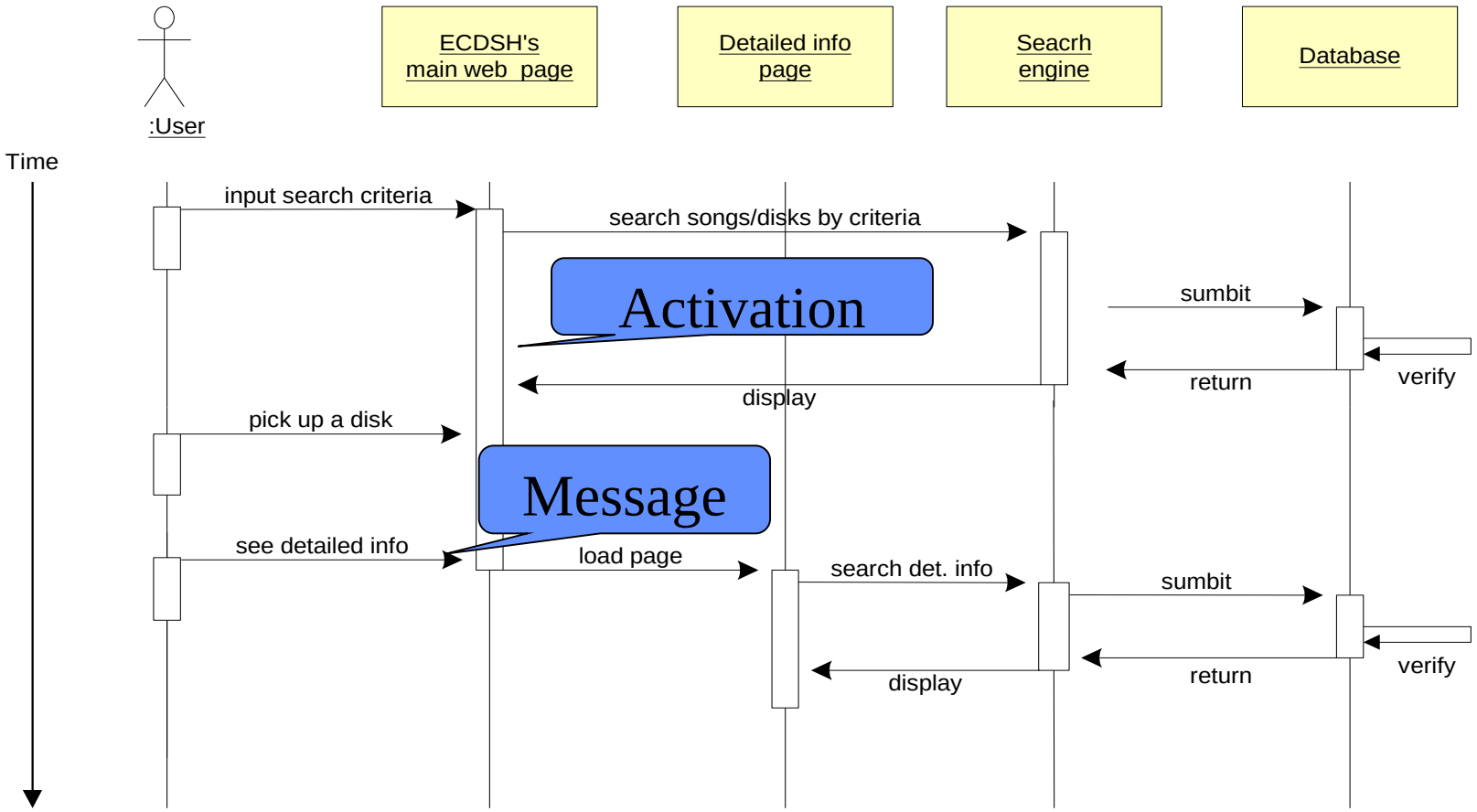
- External “actors”
  - Roles of people
  - Types of systems
- Use cases
  - Top-level functions (solid arrows to/from actors)
- Relationships among use cases
  - Always-depends-on (dashed <<include>>)
  - Sometimes-is-depended-on (dashed <<extend>>)
  - Inherits-from (solid triangle-arrow)

Activity Diagram: Modeling Decisions



Thanks to Satish Mishra

Sequence Diagram



Thanks to Satish Mishra

Good Uses for UML

- Focusing your attention
  - Design from the outside in
- Representing partial understanding
  - Says what you know, silent otherwise
- Validate that understanding
  - Structuring communication with stakeholders

Avoiding UML Pitfalls

- Don’t sweat the notation too much
  - The key is to be clear about what you mean!
- Don’t try to make massive conceptual leaps
  - Leverage encapsulation to support abstraction
- Don’t get to attached to your first design
  - Goal is to find weaknesses in your understanding

Advantages of SCRUM

- Fundamentally iterative, recognizes that requirements change
- Development team in charge of the sprint backlog
  - Favors self-organization rather than top-down control
  - Reprioritize in response to changing requirements and progress
- Time-limited sprints ensure periodic delivery of new product increments
  - Allows opportunities to receive user feedback, change directions, etc.
- Buzzword = *velocity*