Exam #1 Schedule

- Wednesday, Oct 4

Exam #1 Questions

- True/false
- Multiple choice (single/multiple answer choice)
- Fill-in-the-blank
- Discussion

Exam #1 Groundrules

- No cell phones or laptops
- Closed book
- Entire class period will be allowed
- Bring a writing implement
- Remember to bring your Comet ID

Course Topics

- Software development process
- Software requirements engineering
- Architecture & design patterns
- Implementation & coding styles
- Software testing & debugging
- Software refactoring
- Software management

Learning Outcomes

- Ability to
  - understand software lifecycle development models
  - understand and apply software requirements engineering techniques
  - understand and apply software design principles
  - understand and apply software testing techniques
  - understand the use of metrics in software engineering
  - understand formal methods in software development
  - establish and participate in an ethical software development team
  - understand software project management
  - understand CASE tools for software development

understand software lifecycle development models

- Traditional models (waterfall, iterative, prototype)
  - steps of each model
  - pros & cons
  - examples
- Agile
  - agile manifesto: values & principles
  - extreme programming

    core practices

    pros & cons

understand and apply software requirements engineering techniques

- Terminology: stakeholder, functional & non-functional requirements, …
- Requirements engineering process
  - elicitation -> analysis -> specification -> validation
- Elicitation
  - approaches: brainstorm, interviews, ethnography, strawman/prototype
  - terminology: closed/open interviews, …
  - pros & cons
  - combination of different approaches

understand and apply software requirements engineering techniques

- Analysis
  - priority: essential, desirable, optional
- Specification
  - informal -> formal
    - nature language
    - form-base
    - graph notation: use case diagram
    - mathematical
  - pros & cons
  - examples
  - write specification (nature language, form-based, and use case diagram) given informal description

understand and apply software requirements engineering techniques

- Validation & verification
- techniques: review, prototype, testing, verification
- techniques -> process models

  prototyping -> prototype model

  testing -> extreme programming

  verification -> waterfall

understand and apply software design principles

- Object-oriented analysis
  - procedural vs. object-oriented approaches: pros & cons
  - terminology: class, object, life cycle
  - class diagram & sequence diagram

    details: aggregation, composition, multiplicity, lifeline, …

    be ready to draw (remember the notations)
  - OOA tasks and principles

understand and apply software design principles

- Software architecture design
  - motivation & facts
    what? why? who?
  - styles: pipe and filter, layered, MVC, repository
    structural pattern and computational model
    common examples
    pros & cons

understand and apply software design principles

- Design factors and metrics
  - modularity, abstraction
  - component independence
  - fault tolerance
  - metrics

    coupling, cohesion, size

    complexity (cyclomatic metric)

understand and apply software design principles

- Design patterns
  - categories: structural, creational, behavioral
  - Six design patterns (composite, factory, visitor, singleton, template method, proxy)

    motivation (what problem solved, when to apply)

    solution (class diagrams, code examples)

other topics

- Software engineering
  - what?
  - facts: success & failures
  - layered view
- Software repositories
  - terminology: commit, branch, merge, …
  - centralized vs. distributed: pros & cons
  - git basics

other topics

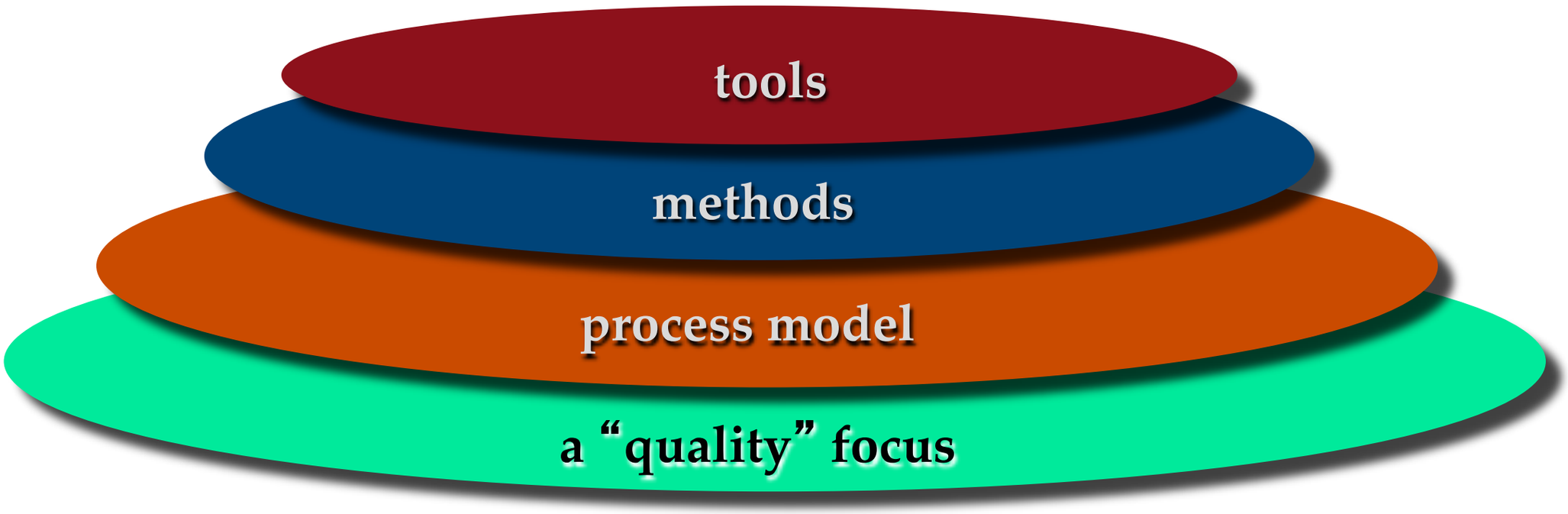
- Coding styles & documentation
  - variable, constant, expression, statement, block, method, file
  - comments

    statements, method, class
  - Javadoc annotations

CE/CS/SE 3354
Software Engineering

Introduction

Software Engineering: A Layered View

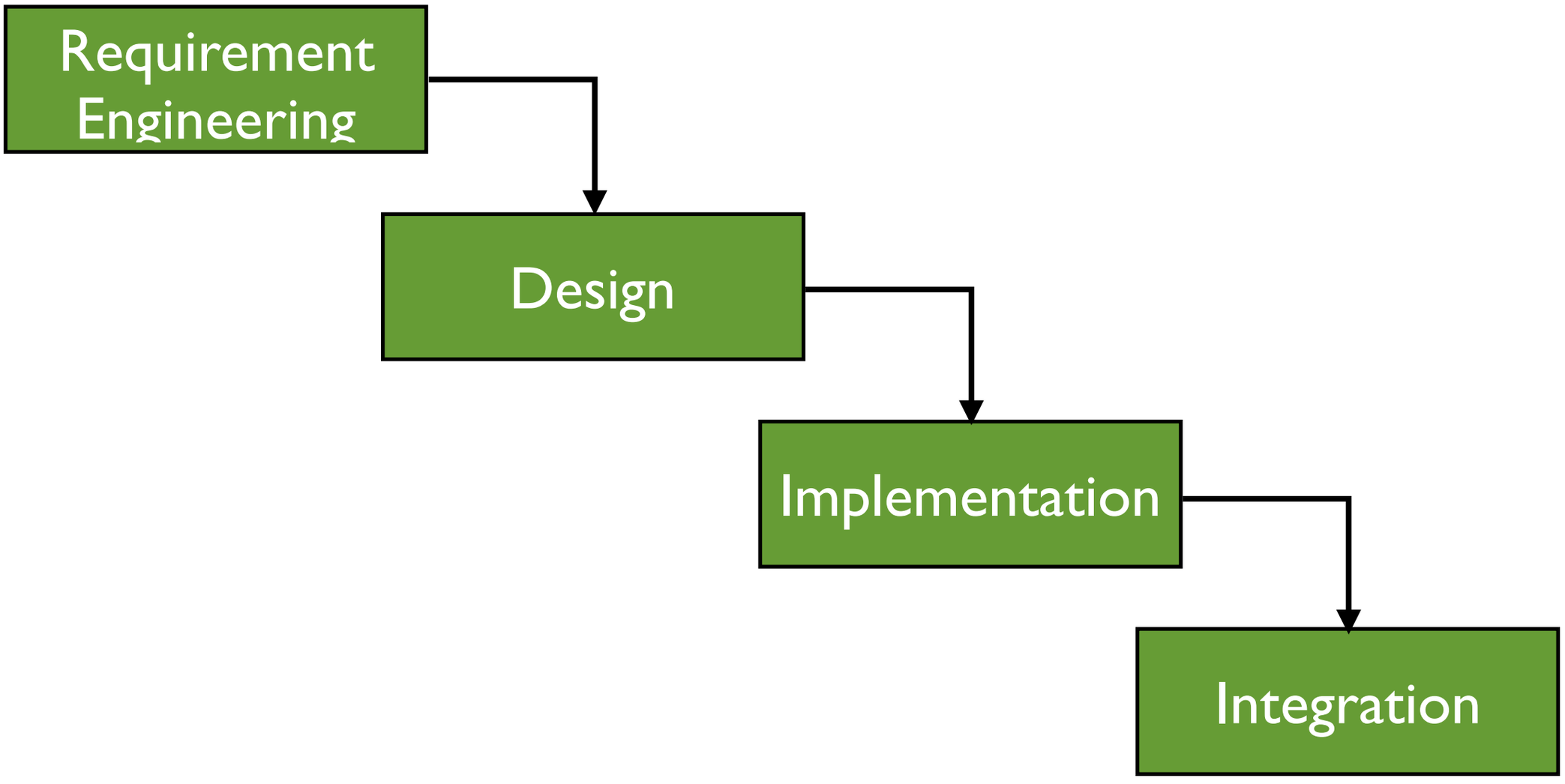


- Tools: provide automated or semi-automated supports for the process and the methods
- Methods: provide "how to's" for building software
- Process: provides a framework for software development

CE/CS/SE 3354 Software Engineering

Software Process Models

The Waterfall Model



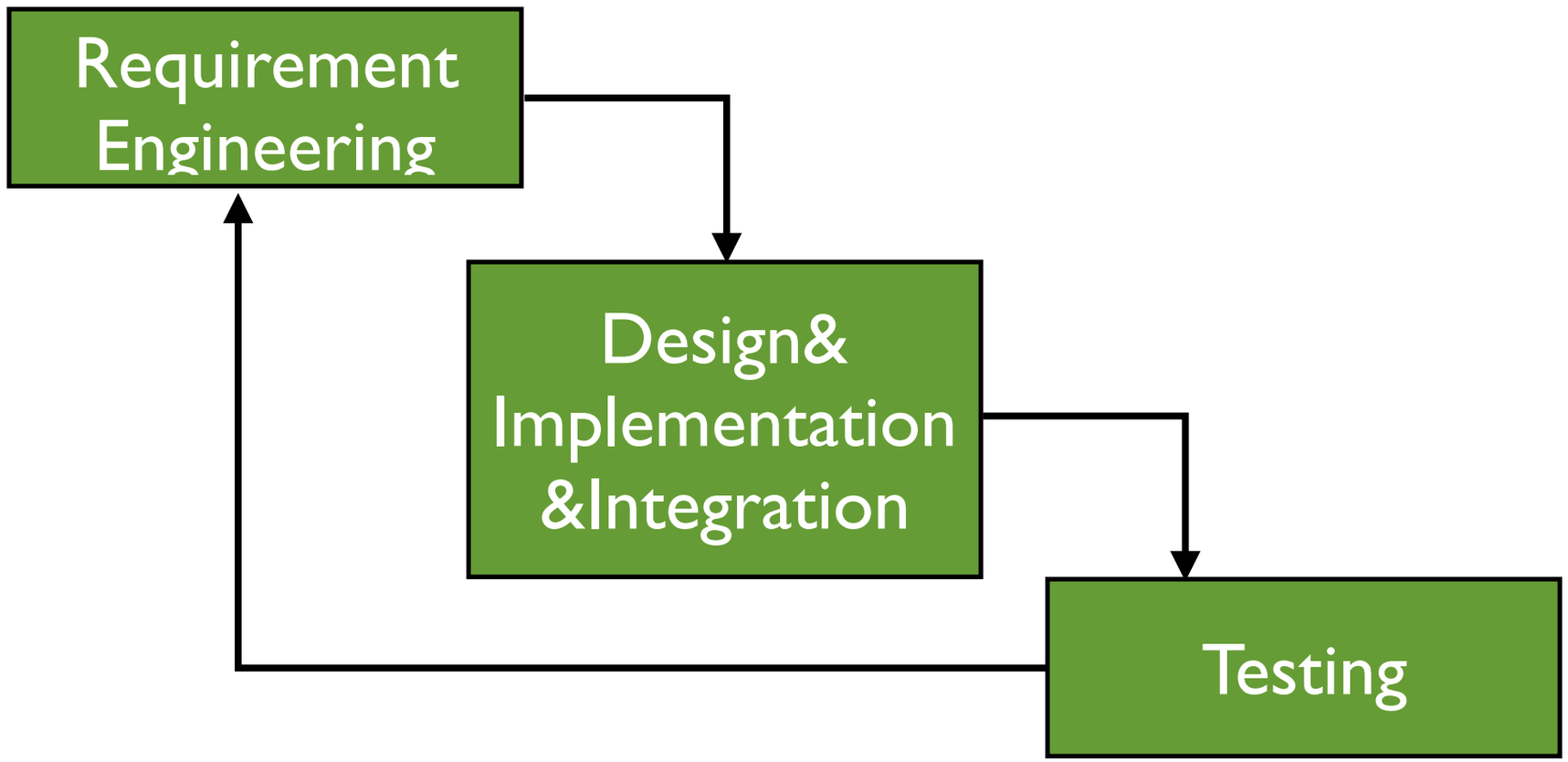
Risks of waterfall

- Relies on precise and stable requirements
- Users cannot involve much (specifications are difficult to understand)
- Takes too long to finish
- Small errors (or requirement changes) at the beginning steps are unaffordable

- Suitable: projects for specific critical software, no competition, enough resources

The Iterative Model

- Solve the risks of waterfall by infinite weak cycles



Advantages

- Users involve earlier
  - User can give feedback after the first version released
- Cheaper to get a working software
  - Get the first version very fast
- The software always work, though not perfect
  - Important in many cases, e.g., in competitions
- Keep refining requirements, and accommodates changes
  - Cost for requirement changes/errors are small

Disadvantages

- More bugs, sometimes may cause severe loss

420k lines of code, 17 errors

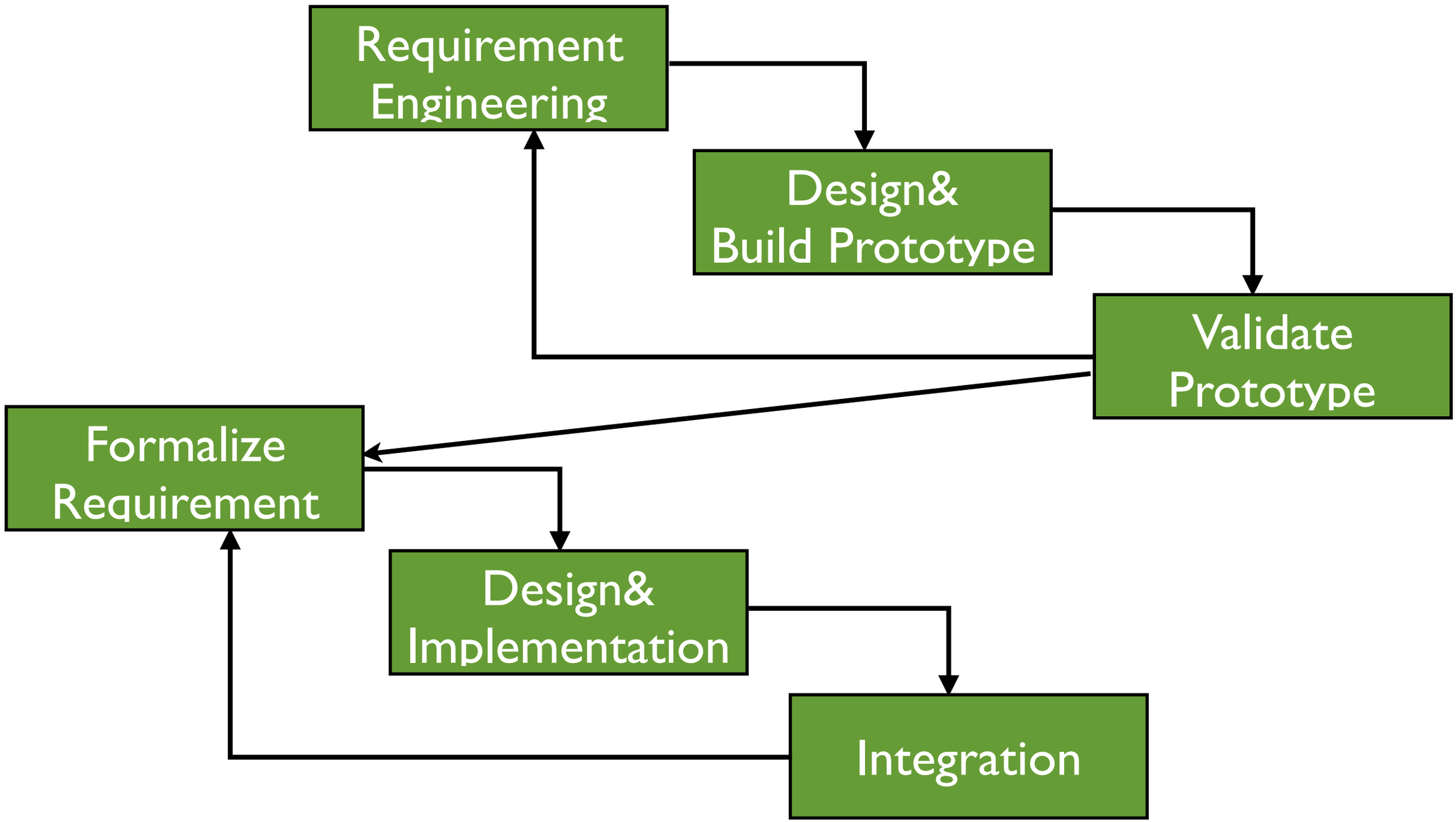
260k lines of code, 5770 bug reports

- Design is critical to ensure that a change does not affect the whole system
  - Ant has 12,000+ code commits

In Practice

- Most existing software projects use this model
  - Daily builds
  - Releases
- Not suitable for systems that are costly for testing or very critical in quality
  - NASA programs
  - Military / Scientific / …

The Prototype Model



The Prototype Model

- Looks like a two-cycle waterfall model
  - Actually not, it is weak + strong
- Different from waterfall
  - Good: users involve more (by using prototype), reveal small errors in requirements / design
  - Not solved: still can not handle frequent requirement changes
  - Bad: prototype is discarded (waste of some effort, sometimes can be even more expensive than waterfall)

XP Core Practices

- Planning
- Small Releases
- System Metaphor
- Simple Design
- Continuous Testing
- Refactoring
- Collective Code Ownership
- Continuous Integration
- 40-hour Work Week
- On-site Customer
- Coding Standards
- Pair Programming

Simple Design

- Just in time
  - Design and implement what you know now, not worry too much about future: future is unpredictable

- No optimization for future
  - It is common that the optimization becomes unnecessary

    Example: optimization to handle large input data, but later found the input changed (e.g., smaller format, or available in a summarized form)

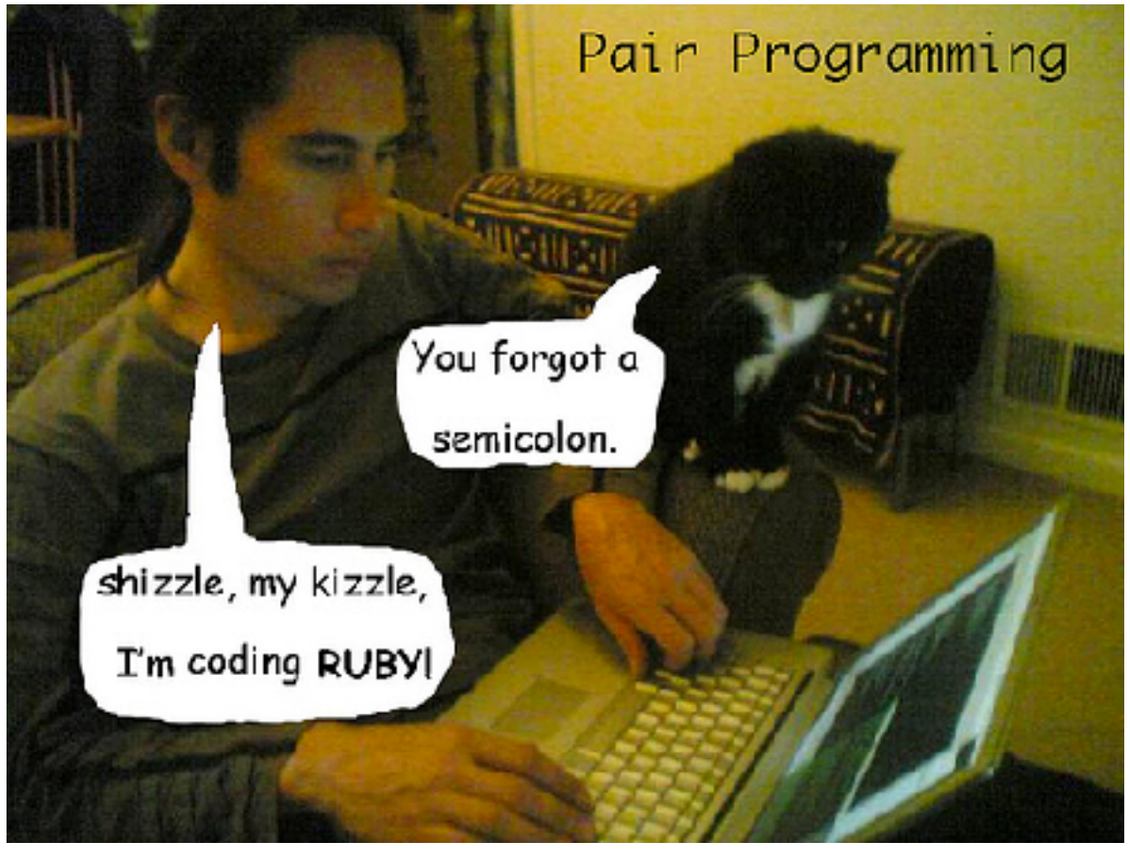
Continuous Testing

- Always keeps code working
  - Test before/after any major changes (Continuous testing)
- Plan coding to allow frequent tests
  - Do not do too comprehensive changes, break to phases

  Example: Add a product query feature for shopping software
    - Add list all products first
    - Add text query
    - Add filtering conditions one by one
    - Add sorting
    - …

Pair Programming

- Programmer and Monitor
  - Pilot and Copilot
  - Or Driver and Navigator
- Programmer types, monitor think about high-level issues
- Disagreement points to design decisions
- Pairs are shuffled



When to use extreme programming

- Requirement prone to change
- Easy to get testable requirements (often true in the maintenance phase on a software)
- Need quick delivery (business competition)

- In practice, frequently used in start-up companies

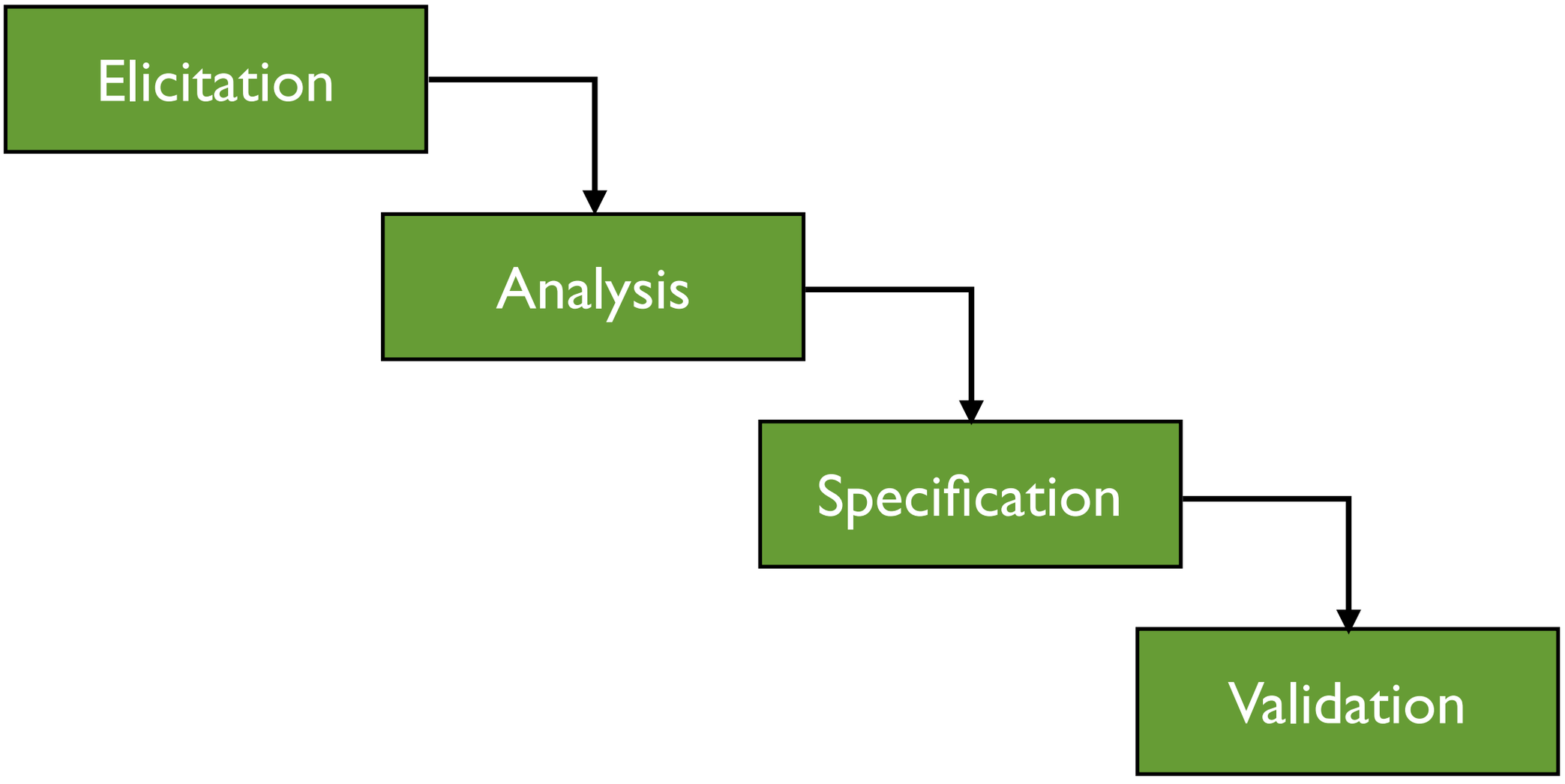
When **not** to use extreme programming

- Quality critical software (e.g., military, NASA projects)
- Large group for large project (still can be used for components)
- No highly-involved customers

CE/CS/SE 3354 Software Engineering

Requirements Engineering

Requirement Engineering Process



Elicitation Approaches

- Brainstorming
- Interviewing
- Ethnography
- Strawman/Prototype

Brainstorm: Pros & Cons?

- Pros
  - No Preliminary Knowledge Preparation
  - Comprehensive gathering of ideas, solve conflicts earlier
- Cons
  - No clear mission, costly for gathering, may take a long time
  - People from different field may feel difficult to interact

Brainstorm Applicability

vs.

- Good: Startup software, general topic, e.g. personal shopping software
- Not good: Domain experts/systems exist, limited resources, e.g., ATM banking system

Interviews: Pros & Cons

- Pros
  - Simple to apply in practice
  - Usually can get some progress

- Cons
  - Interviewee may ignore details because they are too familiar with them
  - Interviewee may have too little knowledge in computer science to express their ideas effectively

Ethnography: Pros & Cons?

- Pros
  - Reveal real requirements, avoid problems caused by imprecision in oral/written expression
  - Require little preliminary knowledge
- Cons
  - May take a long time to finish an effective observation
  - May have legal or privacy issues
  - Can only observe what is happening at present, but people's behavior may change with the new software
- Frequently used in practice when there is an existing system in use

Strawman/Prototype: Pros & Cons

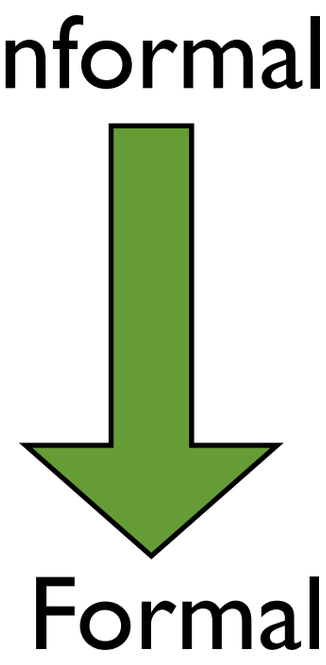
- Pros
  - Can go to details
  - Easy to link requirements to design/implementation
  - Most accurate
- Cons
  - High cost in preparation
  - Require preliminary knowledge
  - Pre-assumptions may limit the scope of the software

Combination of Different Approaches

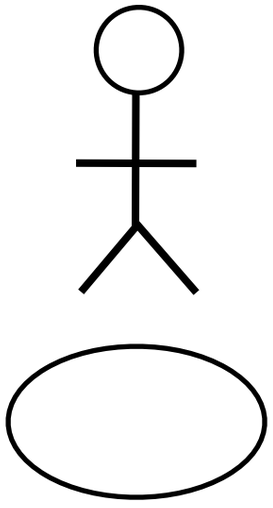
- Brainstorm + interview
  - Raise some questions, then ask more people
- Interview + strawman/prototype
  - Talk to interviewee with a strawman/prototype
- Interview + ethnography
  - Asking people after observing their work
- Prototype + ethnography
  - Observe how people work on a prototype

Requirements Specification

- Natural Language Specification
- Structured Specification
- Graph Notation Specification
- Mathematical Specification



Use Case Diagram Legends



Actor: an entity in the environment that initiates and interacts with the system

Use case: usage of system, a set of sequences of actions

Association: relation between actor and use cases

Includes dependency: a base use case includes the sub use case as a component

Extends dependency: a subtype of use cases that extend the behavior of the base use case

Generalization: one actor can inherit the role of the other actor

CE/CS/SE 3354
Software Engineering

Design

Key steps in OOA

- Define the domain model
  - Find the objects, classes
- Define class diagram
  - Find relationships between classes (static)
- Define the interaction diagrams
  - Describe the interaction between the objects (dynamic)

OOA: Pros

- Code reuse and recycling
- Encapsulation: Objects have the ability to hide certain parts of themselves from programmers
- Design benefits: OO Programs force designers to go through an extensive planning
- Post-implementation benefits: Good design facilitates software maintenance and debugging

OOA: Cons

- Steep learning curve
- Larger program size
- Slower programs
- Not suitable for all types of programs

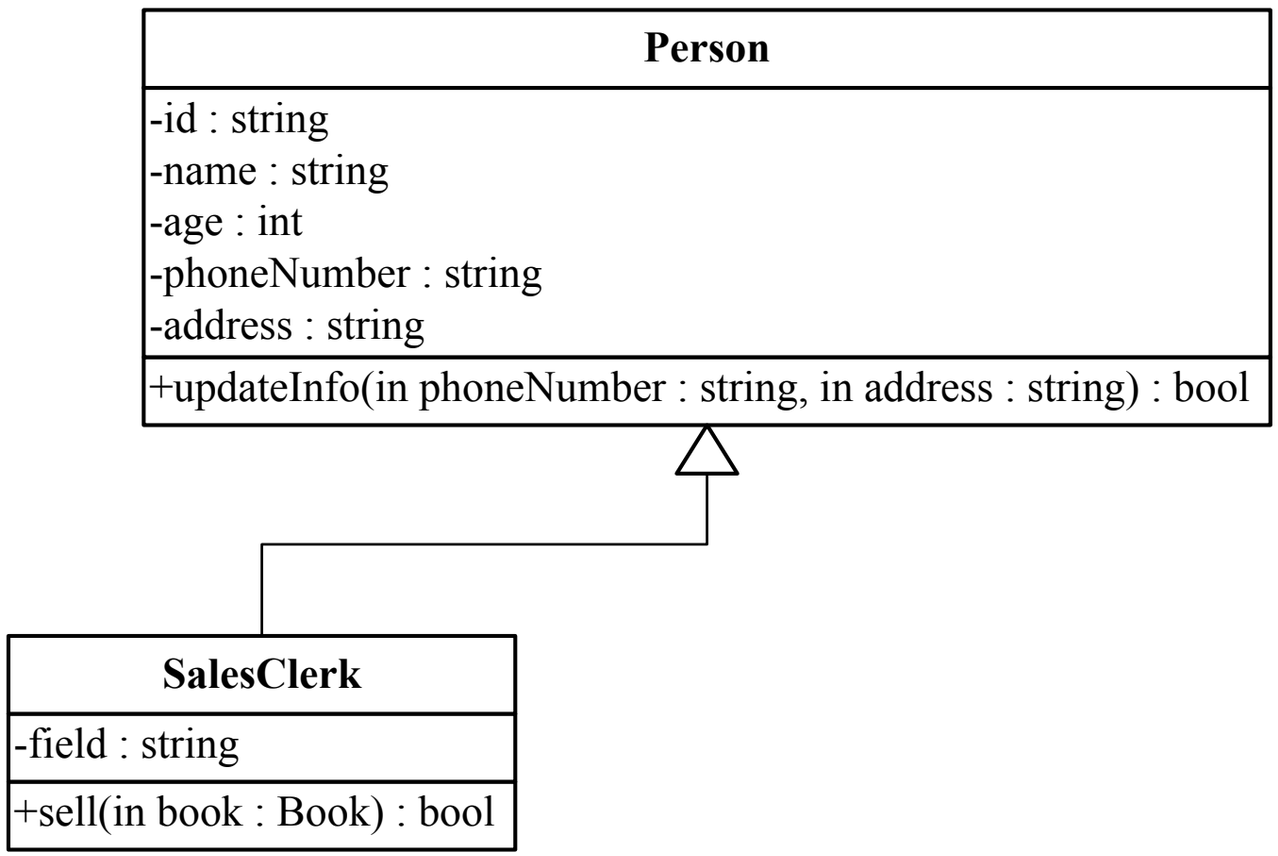
UML Class Diagram Syntax

- Elements of class diagram:
  - Class represented as a box containing three compartments
    - Name
    - Attributes
    - Operations
  - Relation represented as a line between two classes
    - Association
    - Generalization
    - Aggregation and composition

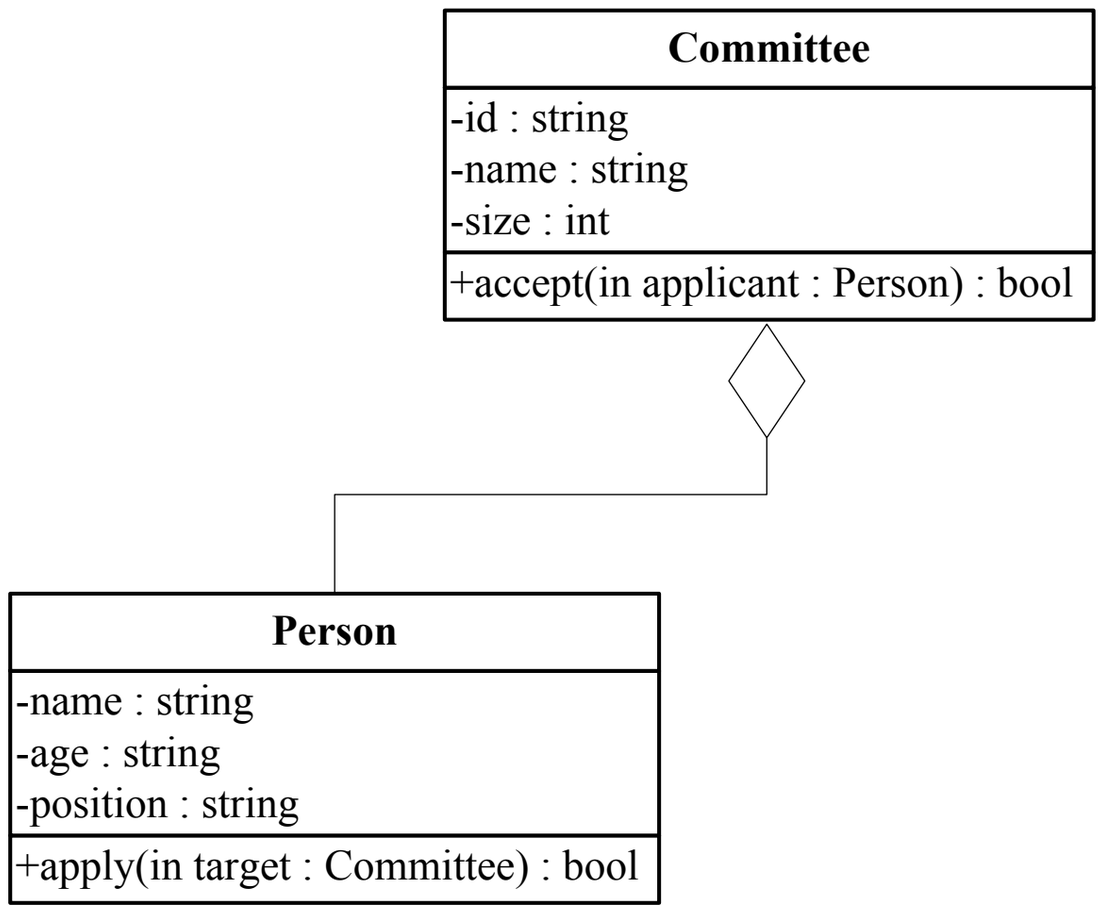
Class Diagram – Class

Shelf
-id : string -size : int -aisle : int -row : int
+loadbook(in book : Book) : bool +removebook(in book : Book) : bool +countbook() : int

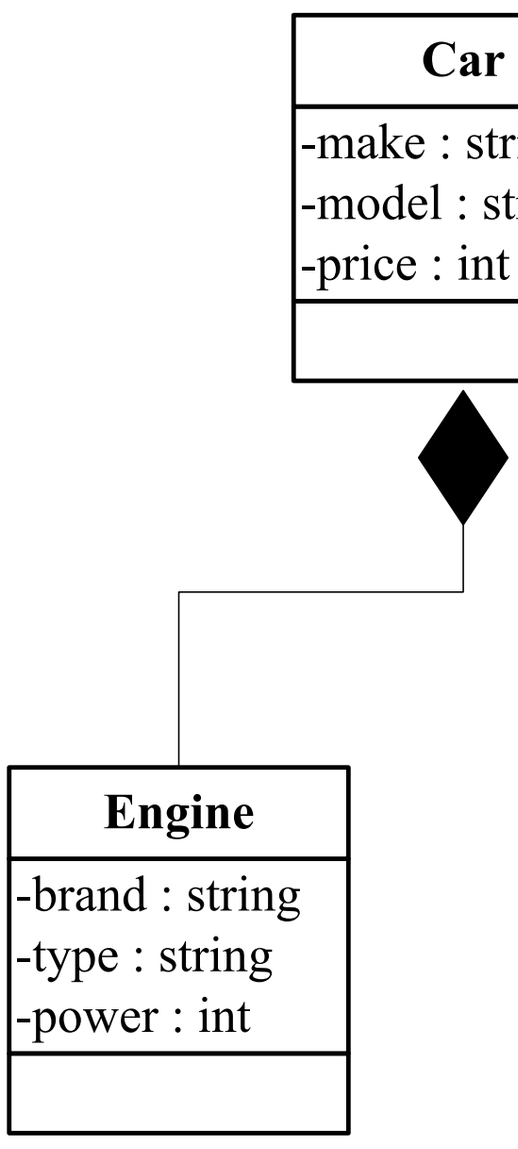
Generalization: example



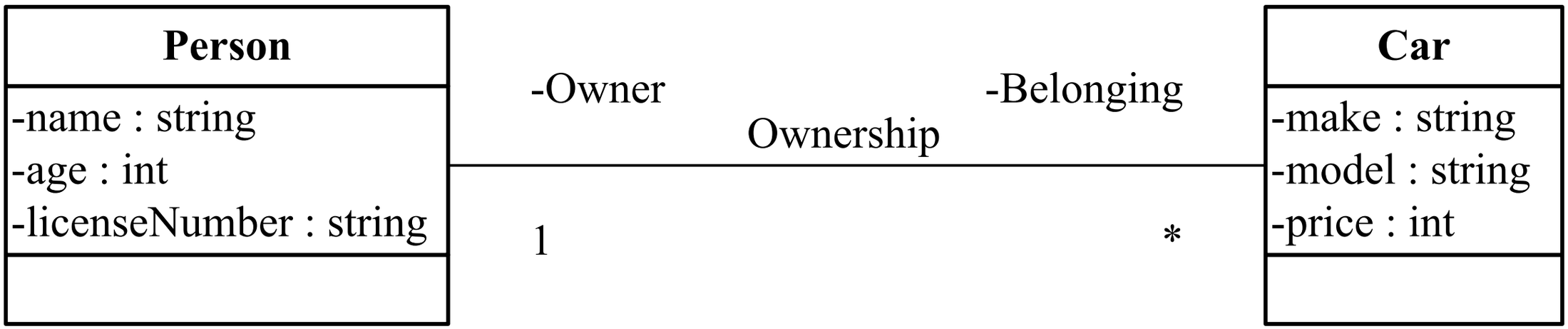
Aggregation Example



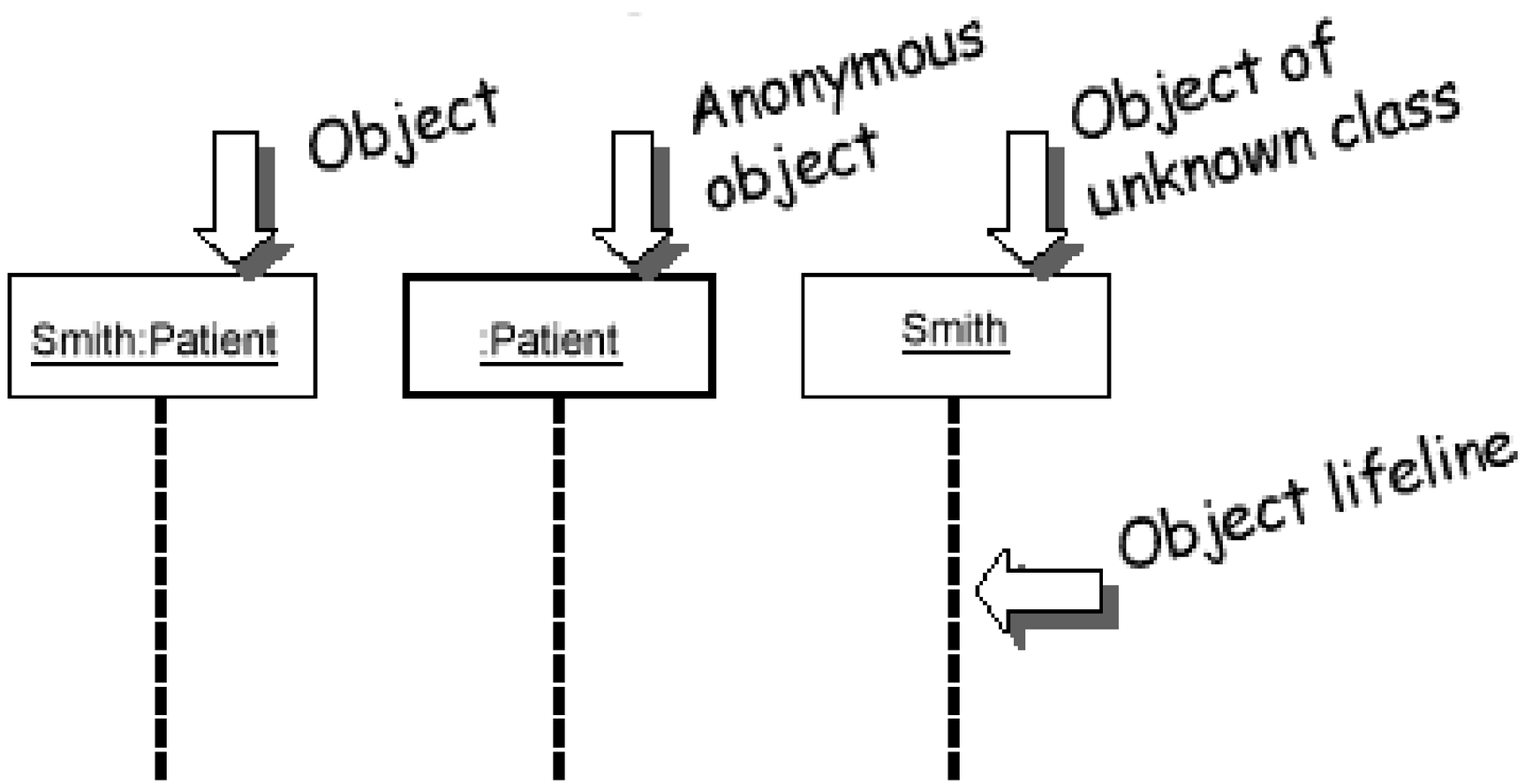
Composition: Example



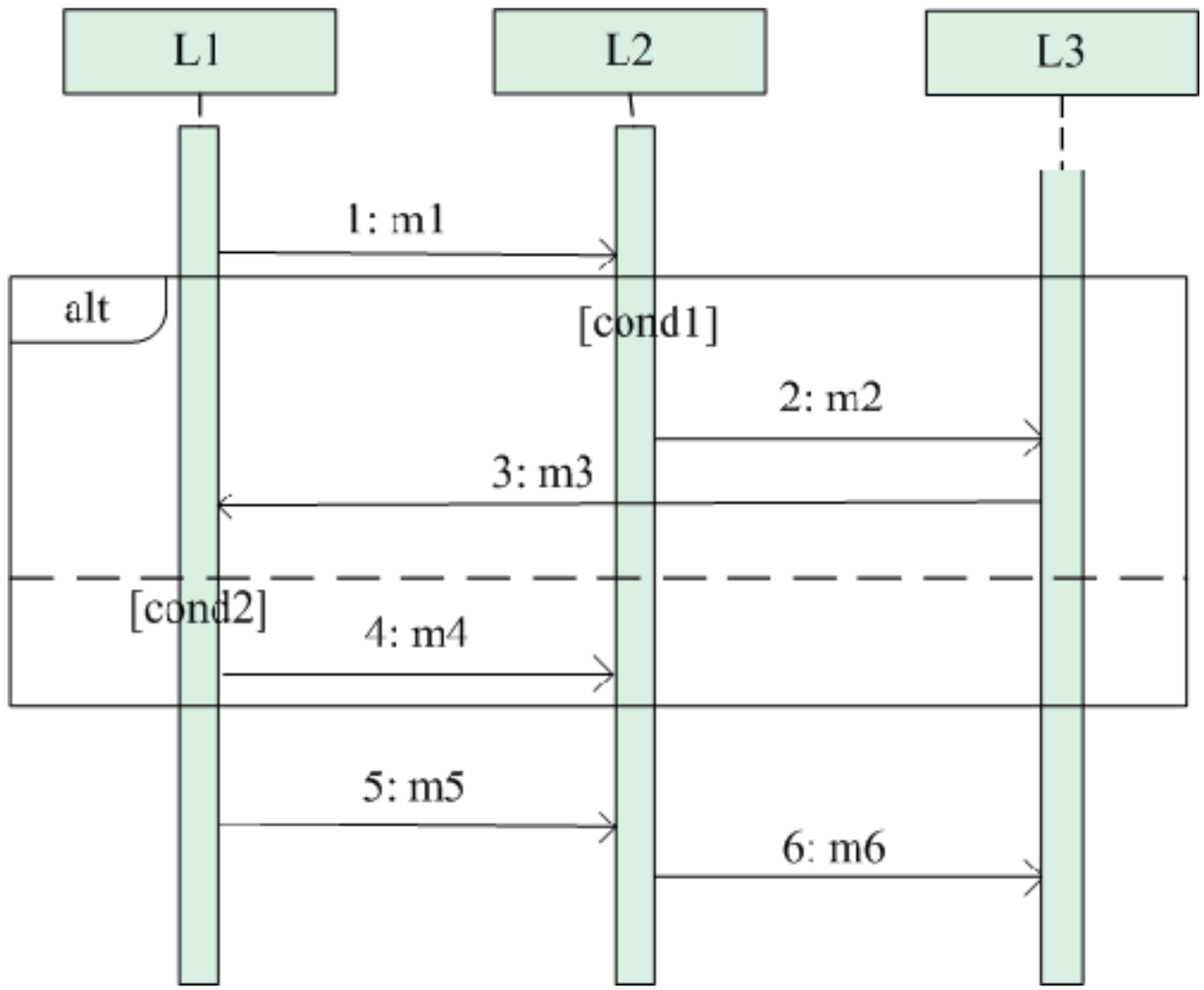
Association Example



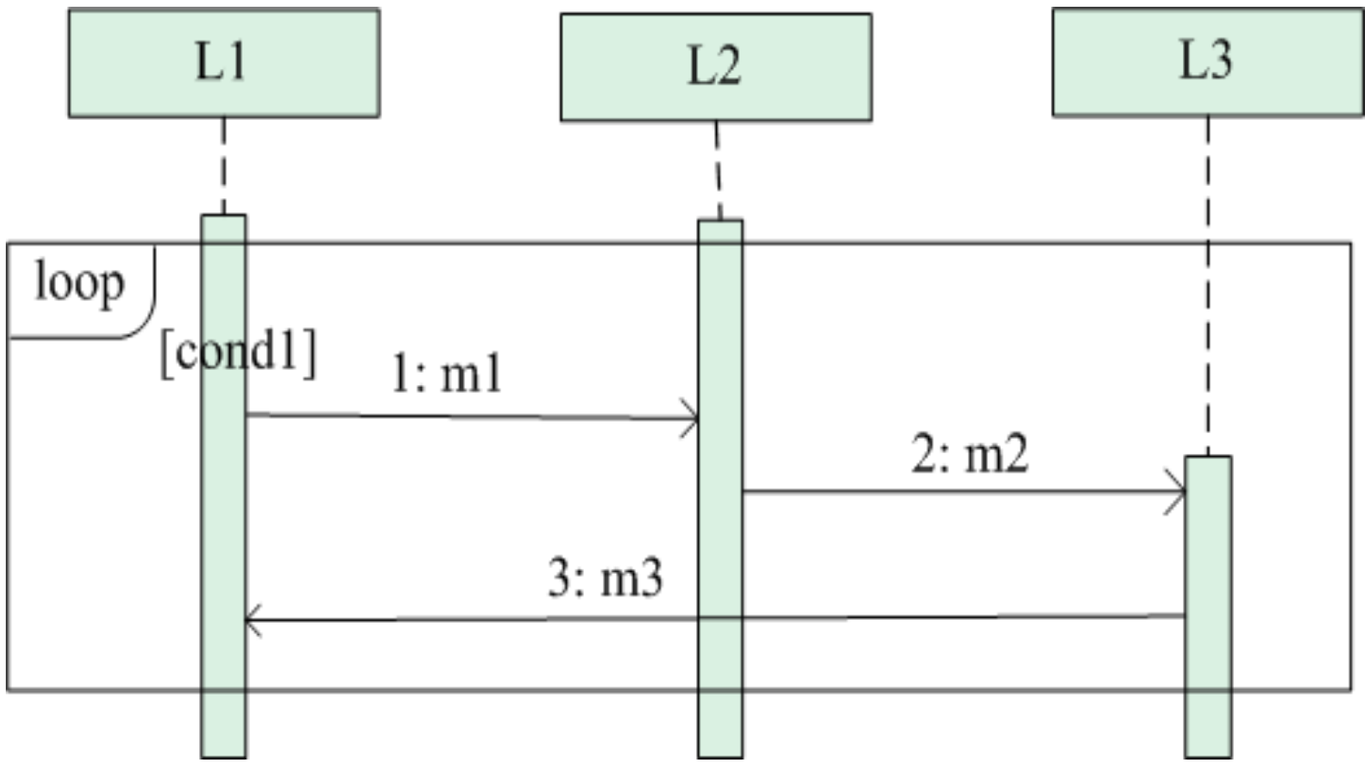
Sequence Diagram: Object and Lifeline



Alternative



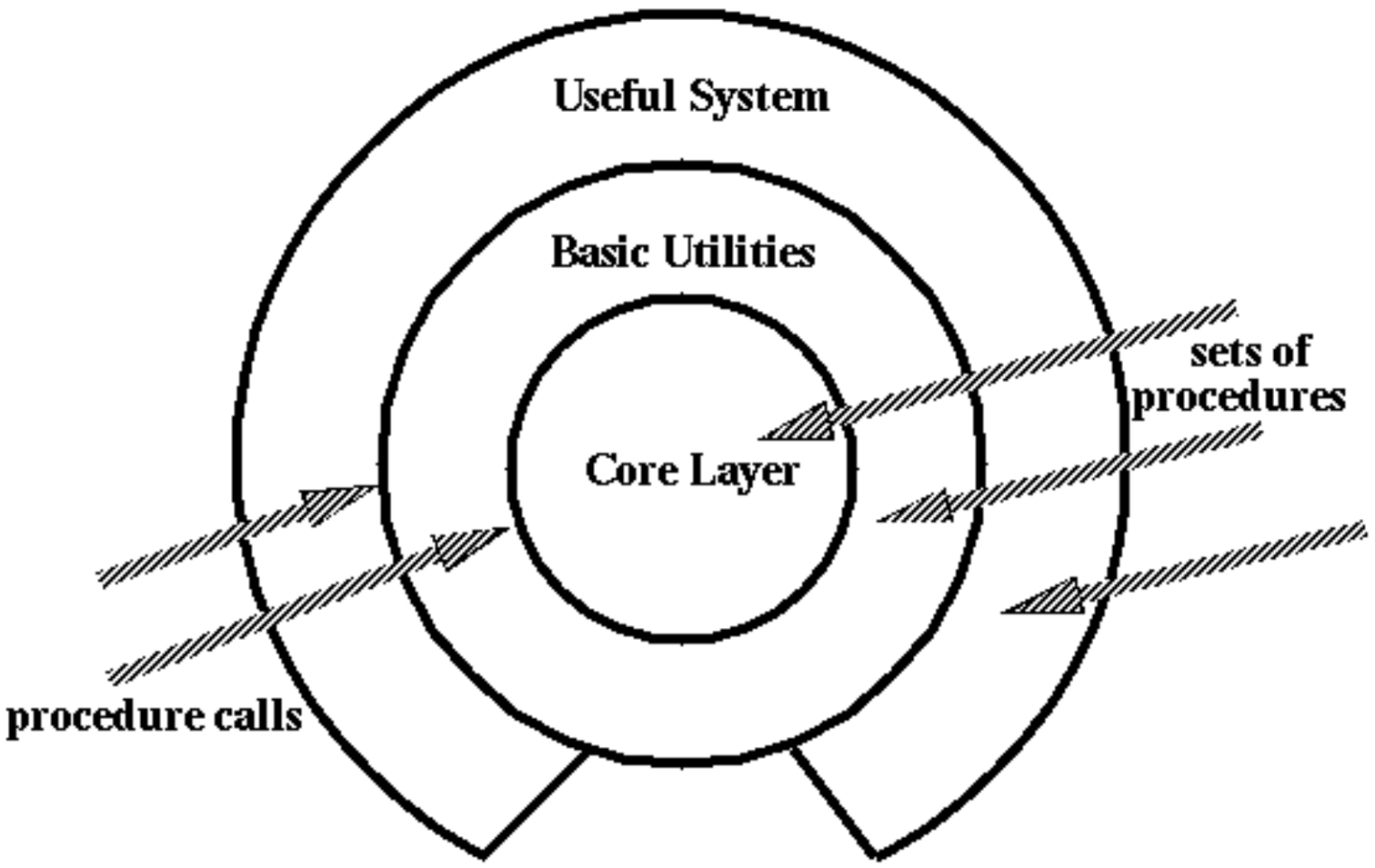
Loop



Popular architecture styles

- Pipe and Filter
- Layered
- Model-View-Controller (MVC)
- Repository

Layered Style



Layered Style: Structure

- Components: are typically collections of procedures
- Connectors: the layer margin, are typically procedure calls under restricted visibility
- Layer Communication Rules:
  - Usually a component will talk only with the layer just beneath it
  - Only carefully selected procedures from the inner layers are made available (exported) to their adjacent outer layer

Layered Style: Examples

- Operating Systems
  - Unix
  - Windows
  - Android
  - …(almost any)
- Distributed Information Systems

Layered Style: Advantages

- Design: based on increasing levels of abstraction
- Security: inner layers are usually not directly accessed by outmost layers
- Maintainability: changes to the function of one layer affects at most two other layers
- Reuse: different implementations (with identical interfaces) of the same layer can be used interchangeably

Layered Style: Disadvantages

- There might be a negative impact on the performance as we have the extra overhead of passing through layers instead of calling a component directly
- The use of layers helps to control and encapsulate the complexity of large applications, but adds complexity to simple applications
- Changes to lower level interfaces tend to percolate to higher levels

Component Independent

- We strive in most designs to make the components independent of one another.
- We measure the degree of component independence using two concepts
  - Low coupling
  - High cohesion

Coupling and Cohesion

- Coupling
  - Two components are highly coupled when there is a great deal of dependence between them
  - Two components are loosely coupled when they have some dependence, but the interconnections among them are weak
  - Two components are uncoupled when they have no interconnections at all
- Cohesion
  - A component is cohesive if the internal parts of the component are related to each other and to its overall purpose

Design Patterns

- Structural Patterns
  - deal with the composition of classes or objects
- Creational Patterns
  - concern the process of object creation
- Behavioral Patterns
  - characterize the ways in which classes or objects interact and distribute responsibility

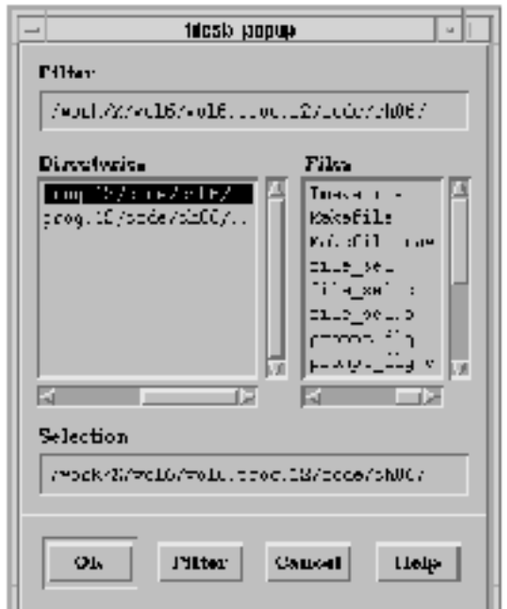
Problem: Supporting Different GUI Styles

- Different GUI styles
  - Appearance of scrollbars, menus, windows, etc., set by the user
  - We want to support them all

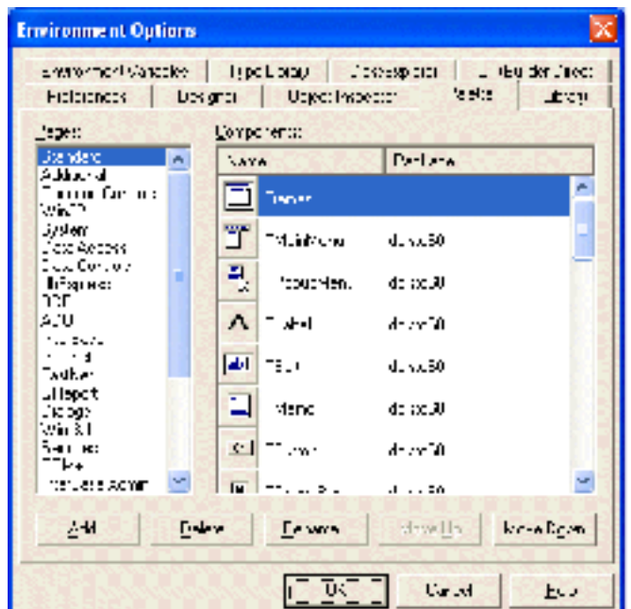
For example: MotifScrollBar and WindowsScrollBar

Both are subclasses of ScrollBars

How should we create a new scrollbar in a window?

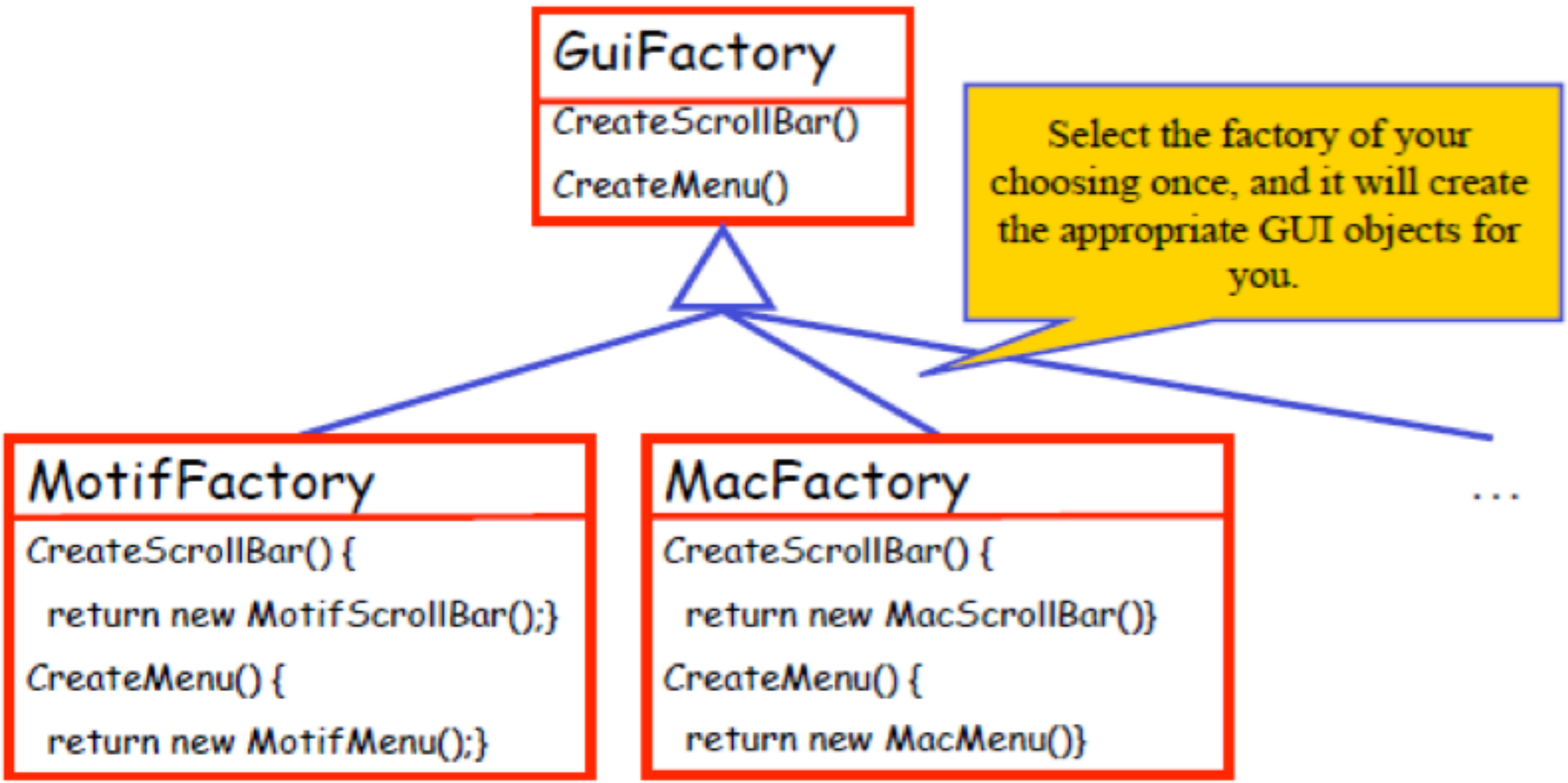


Motif Style



Windows Style

Factory Pattern



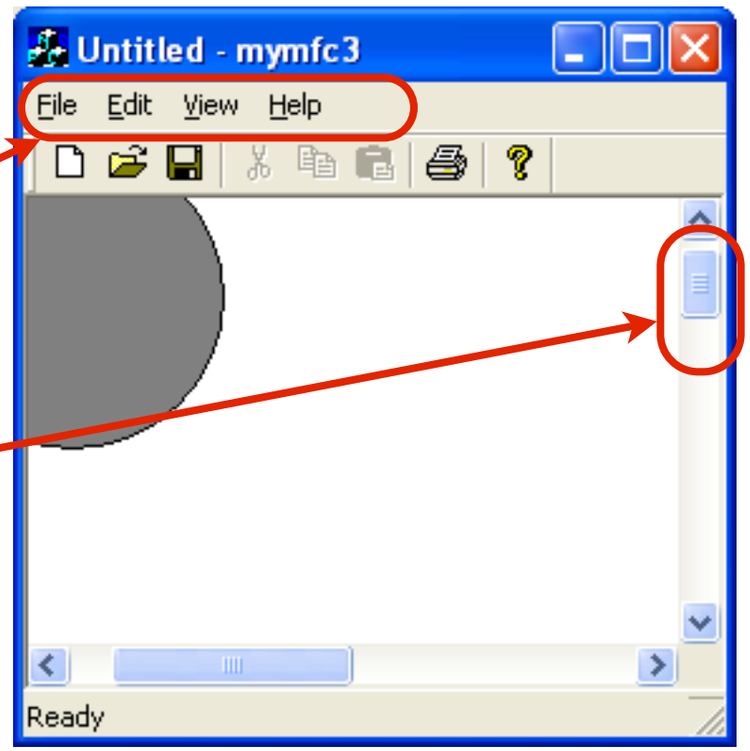
Factory Pattern: Factory Design

```
abstract class GUIFactory{
  abstract ScrollBar CreateScrollBar();
  abstract Menu CreateMenu();
  ...
}

class MotifFactory extends GUIFactory{
  ScrollBar CreateScrollBar(){
     return new MotifScrollBar()
  }
  Menu createMenu(){
     return new MotifMenu();
  }
  ...
}
```

Factory Pattern: GUI code

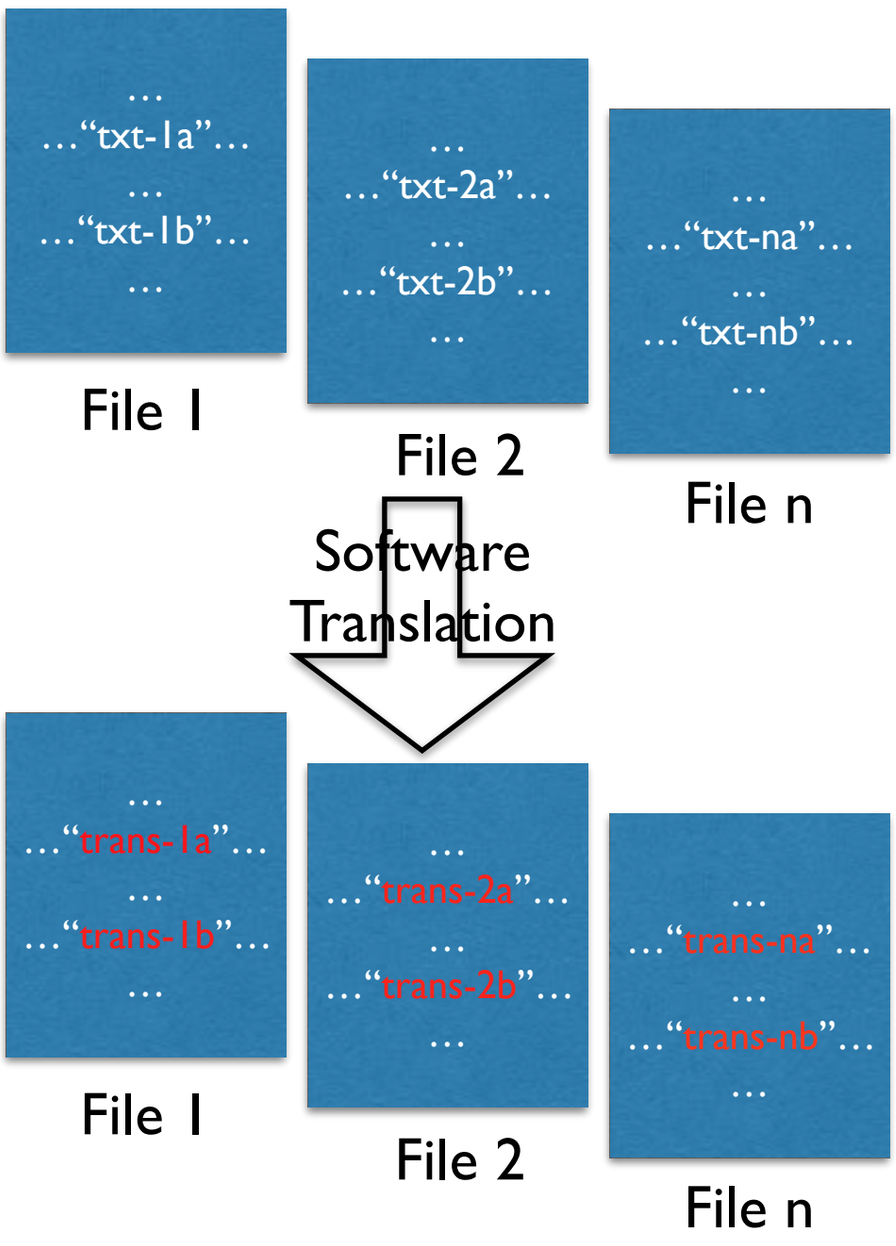
```
GUIFactory factory;
if(style==MOTIF){
    factory = new MotifFactory();
}else if(style==WINDOW){
    factory = new WindowFactory();
}else{
  ...
}
Menu mn = factory.createMenu();
ScrollBar bar = factory.createScrollBar();
```



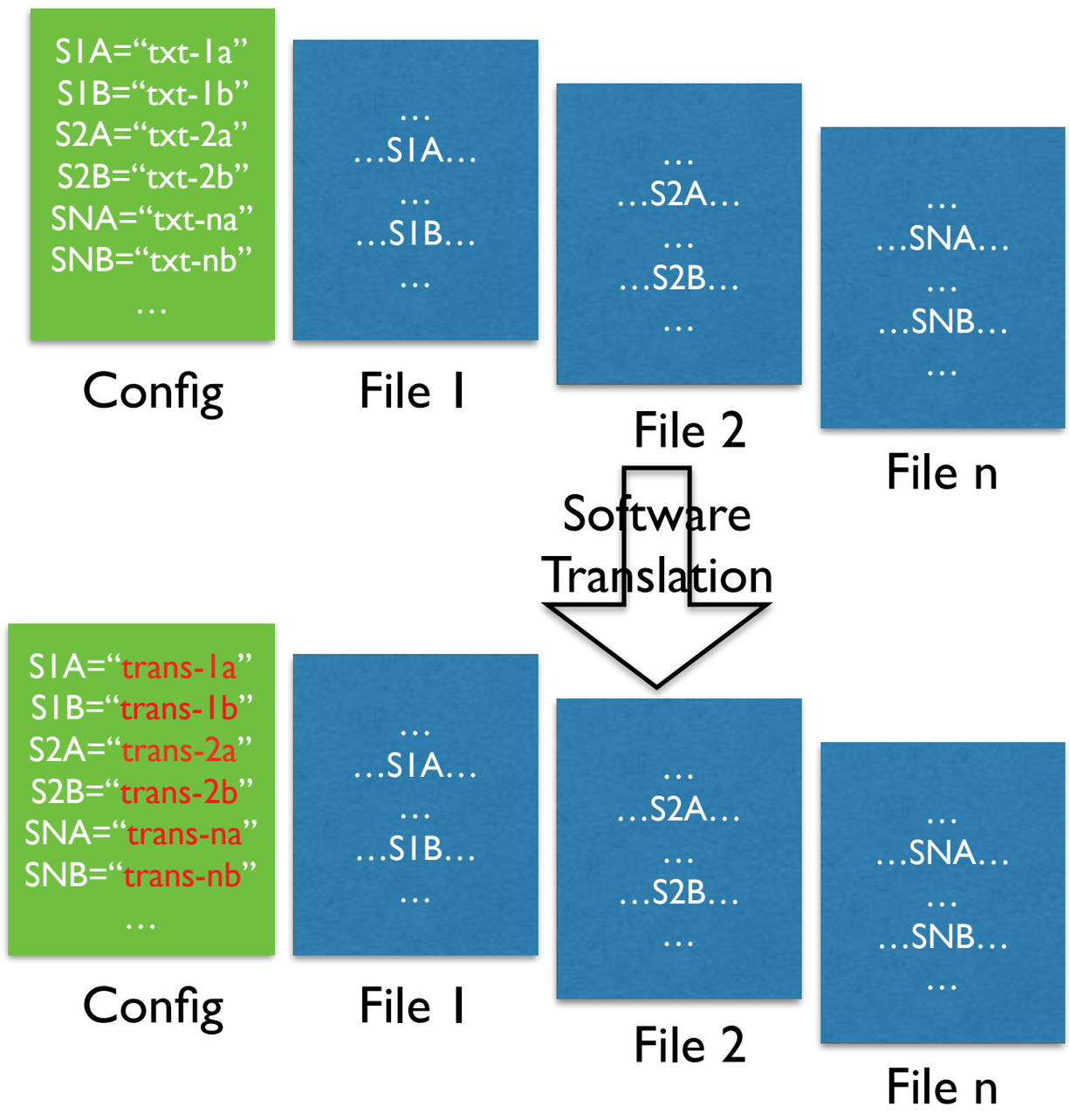
CE/CS/SE 3354
Software Engineering

Coding Styles

String Constants: Example



Bad Style



Good Style

Comments for Methods

- A well-structured format for each public method
- Explain:
  - What the method does
  - What does the return value mean
  - What does the parameters represent
  - What are restrictions on parameters
  - What are the exceptions, and what input will result in exceptions
- For private method, comment may be less structured, but still need to include the information

Comments for Methods

- Example:

```
/**
 * Fetch a sub-array from an item array. The range
 * is specified by the index of the first item to
 * fetch, and ranges to the last item in the array.
 *
 * @param list represents the item list to fetch from
 *                  it should not be null.
 * @param start represents the start index of the
 *                   fetching range it should be between
 *                   0 and the length of list
 * @return the fetched subarray
 */
public List<Item> getRange(List<Item> list, int start){
```