Software

Engineering

The Object Model

何明昕 HE Mingxin, Max

Send your email to c.max@yeah.net with
a subject like: SE-*id-Andy: On What …*

Download from c.program@yeah.net
/文件中心/网盘/SoftwareEngineering2024

Topics

- ✓ Objects and Method Calls
- ✓ Interfaces
- ✓ UML Notations
- ✓ Class/Object Relationships
- ✓ Process/Algorithm-Oriented vs. Object Oriented Approaches
- ✓ How To Design Well OO Systems?

Exercise 03: Review & Preview

- Read 1.3,1.4, 2.1,2.2 and Preview 2.3, 2.4 of TextBook.
- Preview Appendix G (p449-476) of TextBook

- Keep Reading, continue to write work log (just reading jobs in this week) something like:

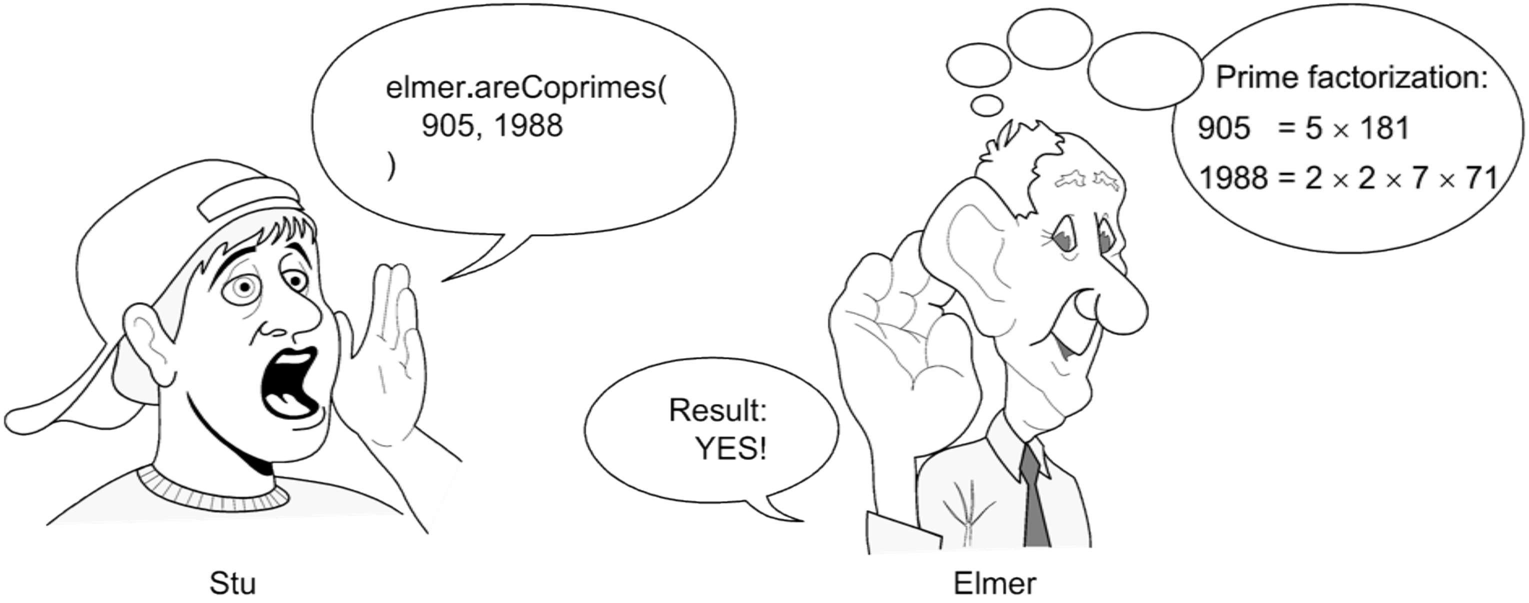
  Read 1.3, 1.4 of TextBook, Mar. 14, 8:00pm-9:20pm

  Read 2.1 of TextBook, Mar. 15, 9:30pm-10:30pm

  Read 2.2 of TextBook, …

  Preview Appendix G (p449-476) of TextBook, DATE, FROM-TO

Objects, Calling & Answering Calls



Prime factorization of 905:

5×181 (2 distinct factors)

Prime factorization of 1988:

2×2×7×71 (4 factors, 3 distinct)

Two integers are said to be coprimes or relatively prime if they have no common factor other than 1 or, equivalently, if their greatest common divisor is 1.

Object

In Computer Science, an Object is a software component that has an identity, stores data that is usually carefully hidden ("private") and provide services. The object provides services through messages, which are the method calls. Of course, to send a message you must know whom to send it, hence the identity, called "reference" in Java.

Identity

Object Data

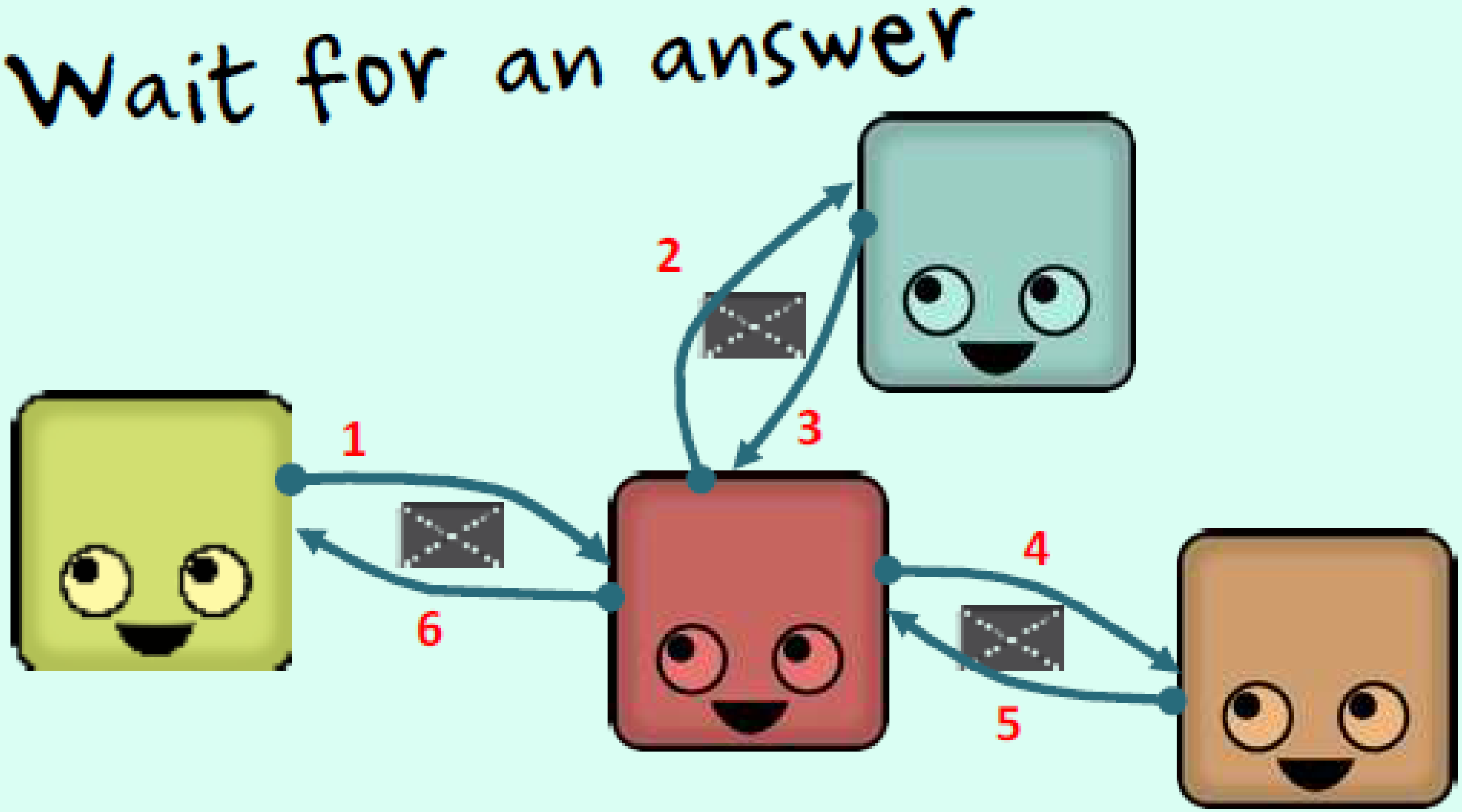
Provides services

message

An object can provide services to oneself. In that case the identity is "this", which simply means "me, this object".

When messages are exchanged, they are synchronous (a word that comes from Greek and means "same time"), which practically means that the caller waits for the answer and does nothing until it gets it.

Mostly **synchronous** communications

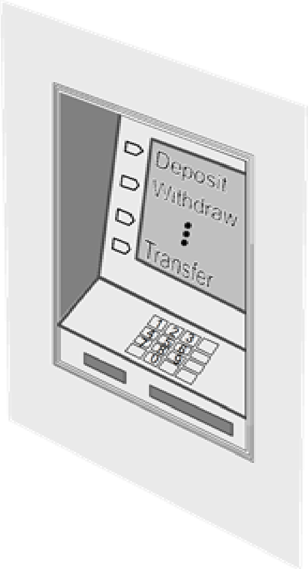


In this example the yellow object sends a message (1) to the red object and is blocked until it gets the answer back (6)

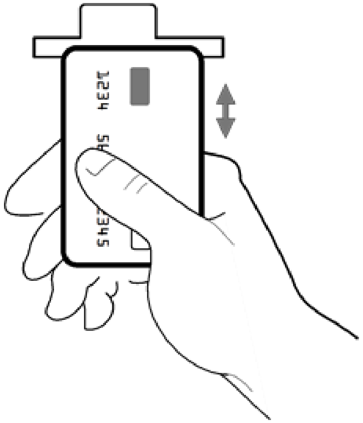
Objects Don’t Accept Arbitrary Calls

Acceptable calls are defined by object “**methods**”
(a.k.a. Operations, Procedures, Subroutines, Functions)

Object:
ATM machine

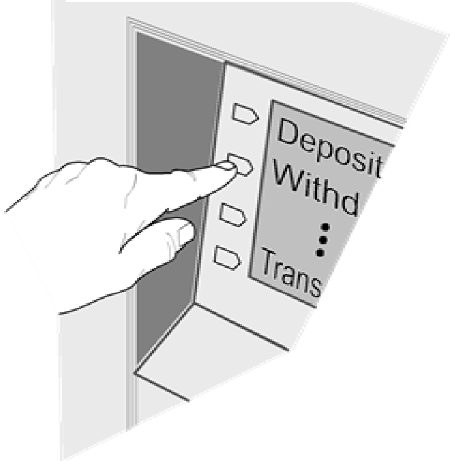


method-1:
Accept card



method-2:
Read code

method-3:
Take selection

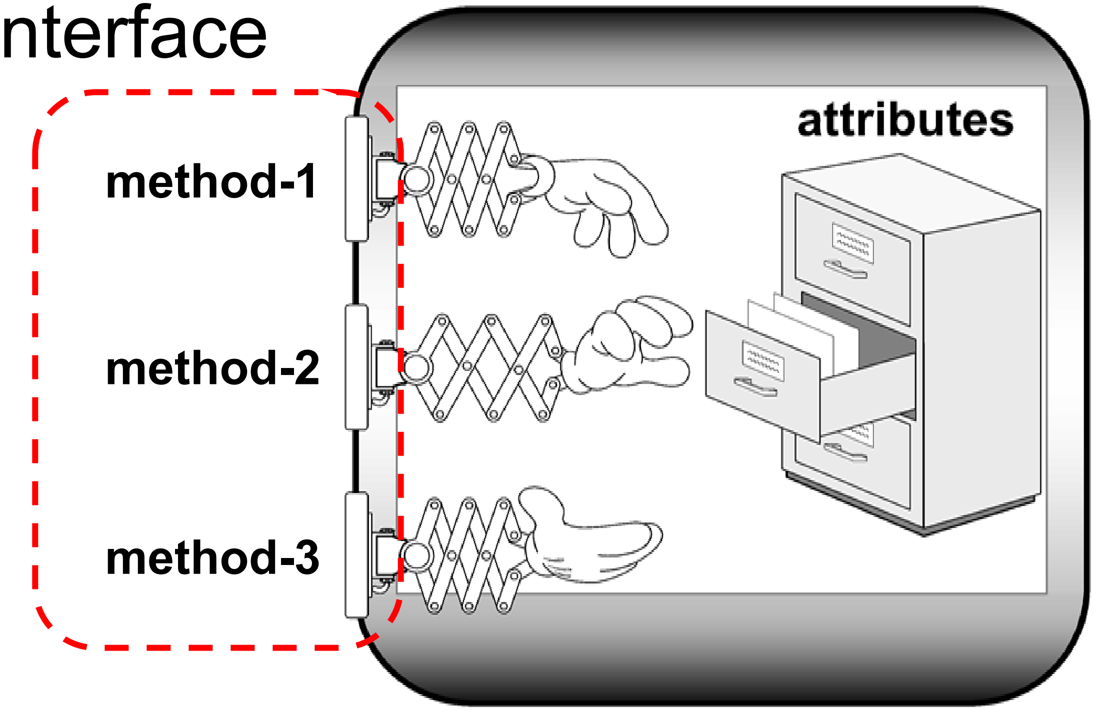


Object Interface

Interface defines method "signatures"

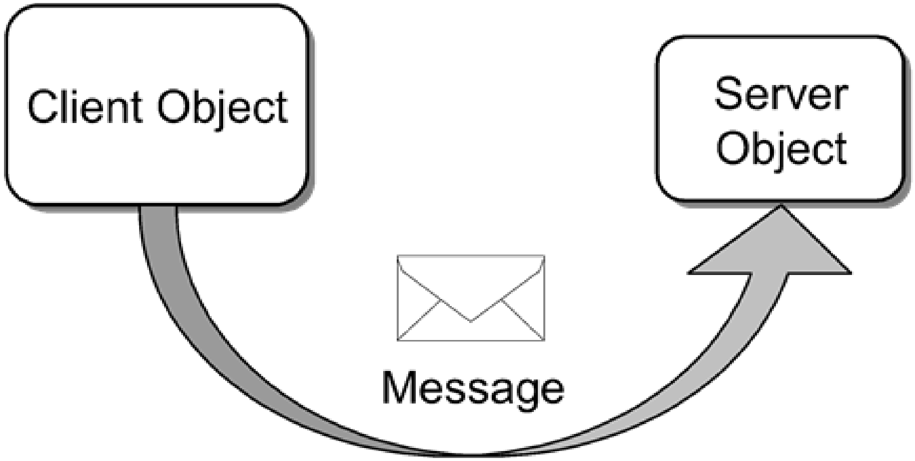
Method signature: name, parameters, parameter types, return type

Interface



Object **hides** its state (attributes). The attributes are accessible only through the interface.

Clients, Servers, Messages



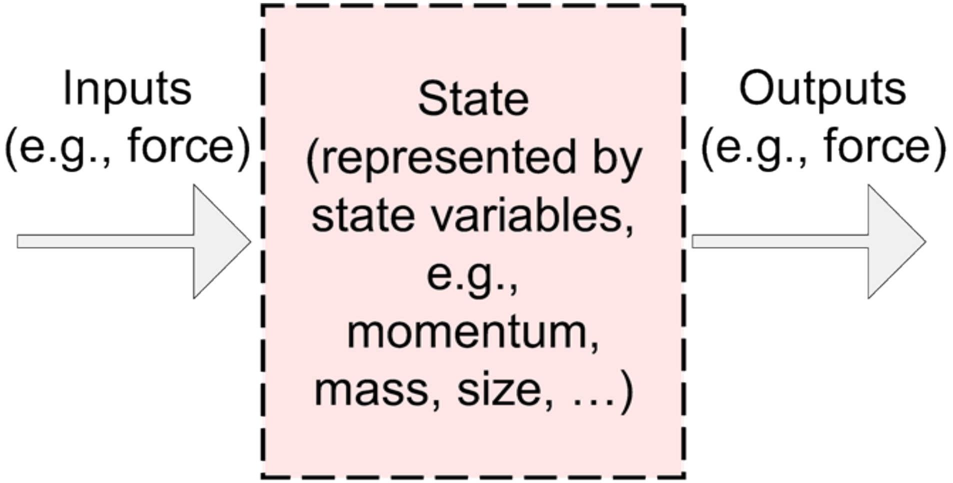
- Objects send **messages** by calling methods
- **Client object**: sends message and asks for service
- **Server object**: provides service” and returns result

Interfaces

- An interface is a set of functional properties (services) that a software object provides or requires.
- Methods define the “services” the server object implementing the interface will offer
- The methods (services) should be created and named based on the needs of client objects that will use the services
  - “On-demand” Design — we “pull” interfaces and their implementations into existence from the needs of the client, rather than “pushing” out the features that we think a class should provide

Objects are Modules

Software Module



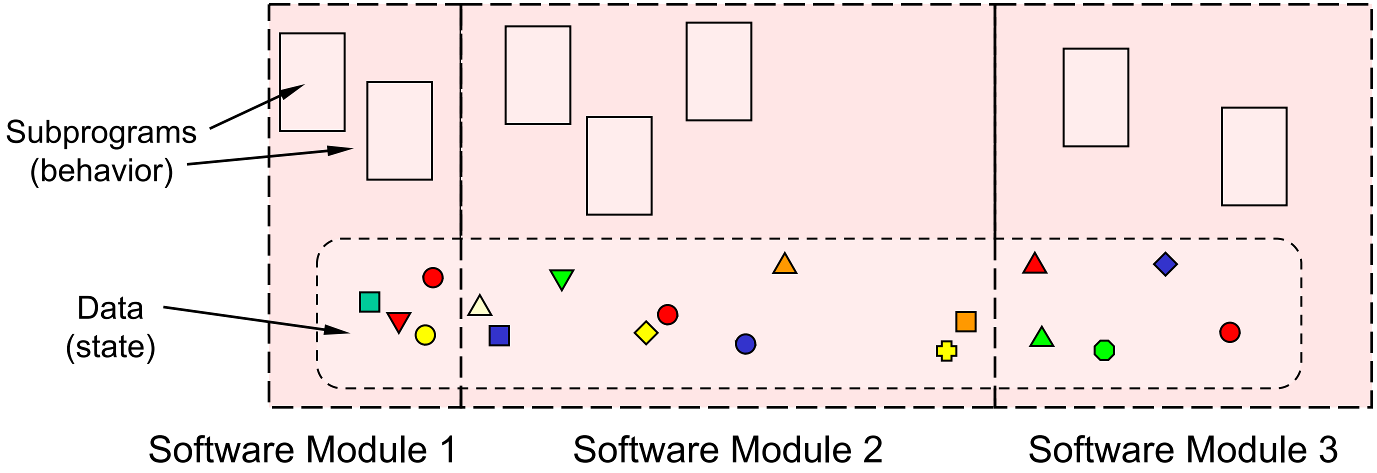
Inputs
(e.g., force)

State
(represented by
state variables,
e.g.,
momentum,
mass, size, …)

Outputs
(e.g., force)

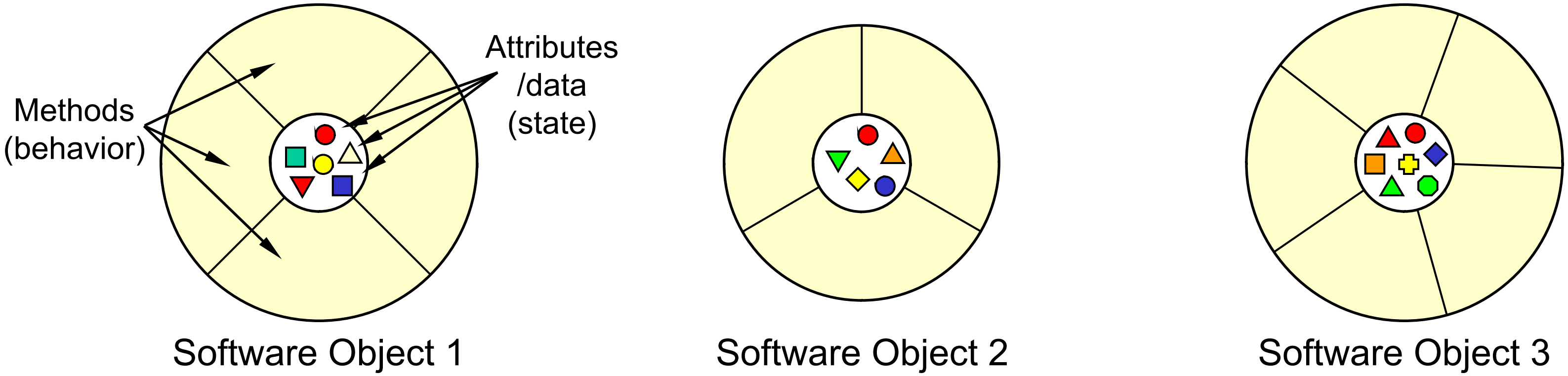
Modules versus Objects

Modules are loose groupings of subprograms and data



"Promiscuous" access to data often results in misuse

Objects *encapsulate* data



UML Diagrams

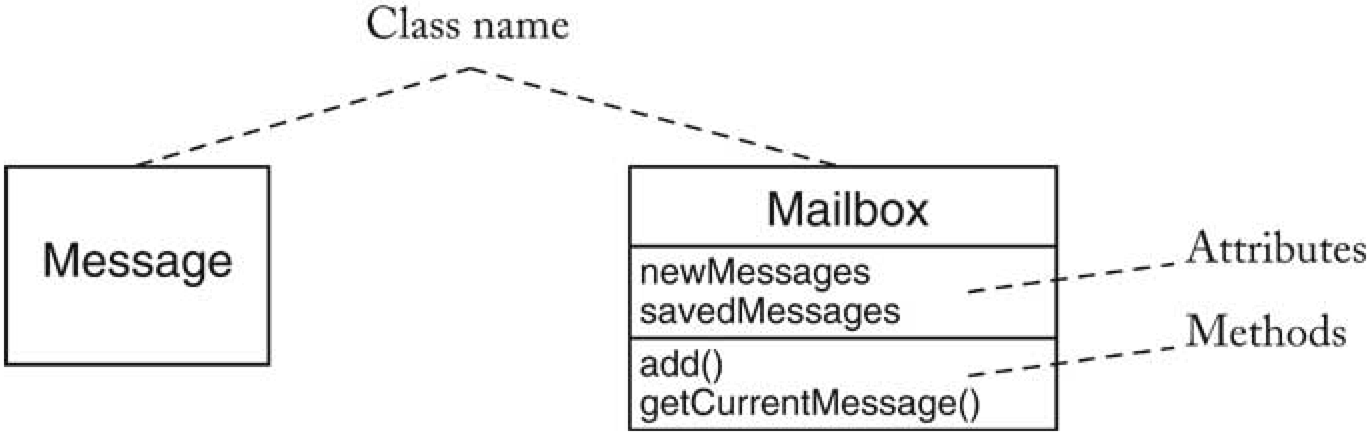
http://www.uml.org

- UML = Unified Modeling Language
- Unifies notations developed by the "3 Amigos" Booch, Rumbaugh, Jacobson
- Many diagram types
- Diagrams used most:
  - Class Diagrams
  - Sequence Diagrams
  - State Diagrams

Class Diagrams

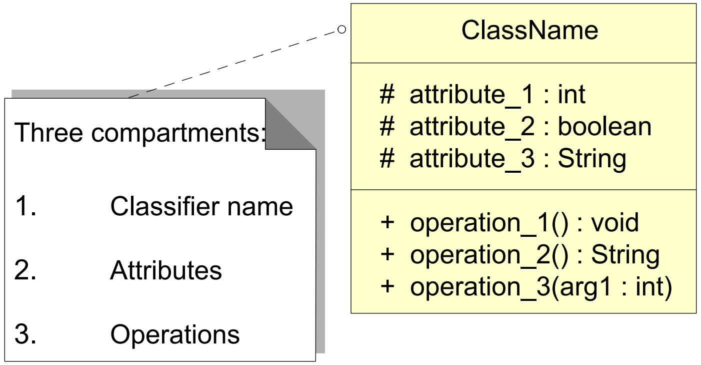
- Rectangle with class name
- Optional compartments
  - Attributes
  - Methods
- Include only key attributes and methods

Class Diagrams



UML Notation for Classes

Software Class



Three compartments:

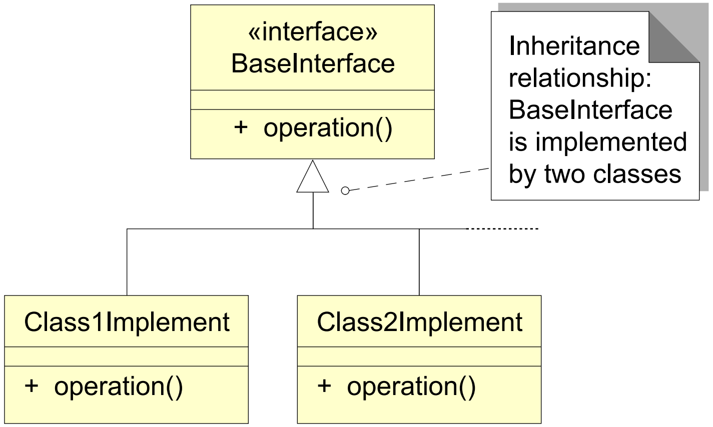
1. Classifier name
2. Attributes
3. Operations

ClassName

# attribute_1 : int
# attribute_2 : boolean
# attribute_3 : String

+ operation_1() : void
+ operation_2() : String
+ operation_3(arg1 : int)

Software Interface Implementation
(traditional notation)



«interface»
BaseInterface

+ operation()

Inheritance relationship: BaseInterface is implemented by two classes

Class1Implement

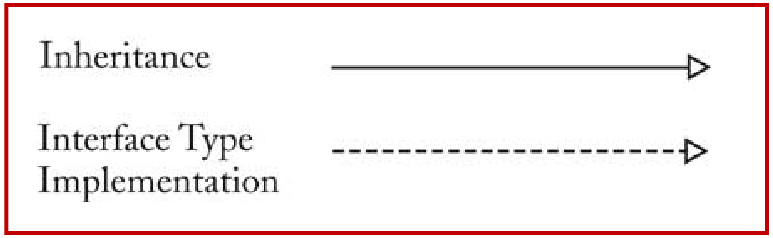
+ operation()

Class2Implement

+ operation()

Access Modifiers (Visibility):

+ public
# protected (package + sub-classes)
~ package
- private



Inheritance

Interface Type Implementation

Class Relationships

- Dependency ("uses")
- Aggregation ("has")
- Inheritance ("is")

Dependency Relationship

- `C` depends on `D`: Method of `C` manipulates objects of `D`
- Example: `Mailbox` depends on `Message`
- If `C` *doesn't use* `D`,
  then `C` can be developed without knowing about `D`

Coupling

- Minimize dependency: reduce *coupling*
- Example: Replace

  ```
  void print() // prints to System.out
  ```

  with

  ```
  String getText() // can print anywhere
  ```

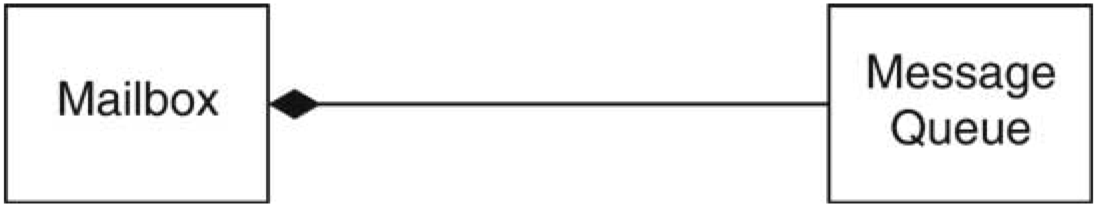
- Removes dependence on `System`, `PrintStream`

Aggregation

- Object of a class contains objects of another class
- Example: `MessageQueue` aggregates `Messages`
- Example: `Mailbox` aggregates `MessageQueue`
- Implemented through instance fields

Composition

- Special form of aggregation
- Contained objects don't exist outside container
- Example: message queues permanently contained in mail box

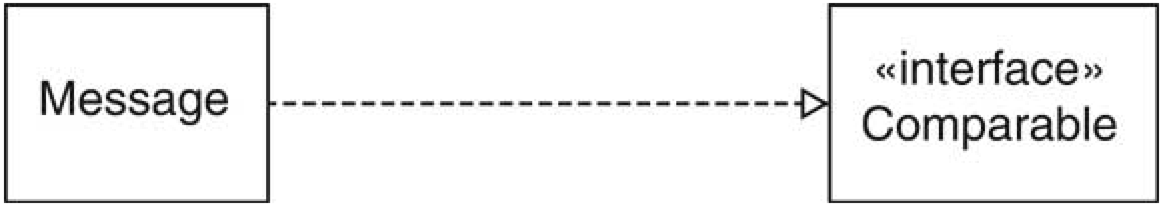


Inheritance (Generalization/Specialization)

- More general class = superclass
- More specialized class = subclass
- Subclass supports all method interfaces of superclass (but implementations may differ)
- Subclass may have added methods, added state
- Subclass inherits from superclass
- Example: `ForwardedMessage` inherits from `Message`
- Example: `Greeting` *does not* inherit from `Message` (Can't store greetings in mailbox)

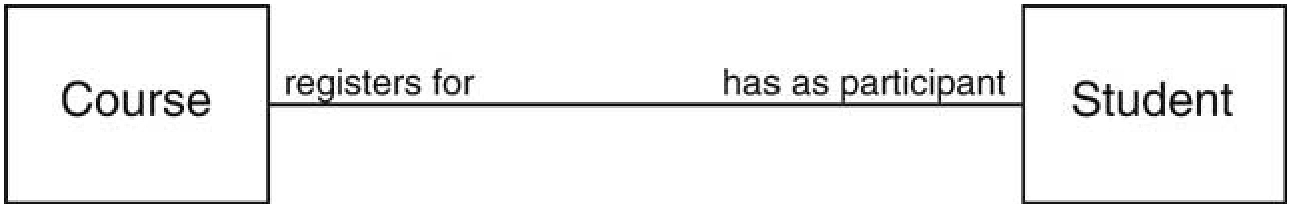
Interface Types

- Interface type describes a set of methods
- No implementation, no state
- Class implements interface if it implements its methods
- In UML, use stereotype «interface»



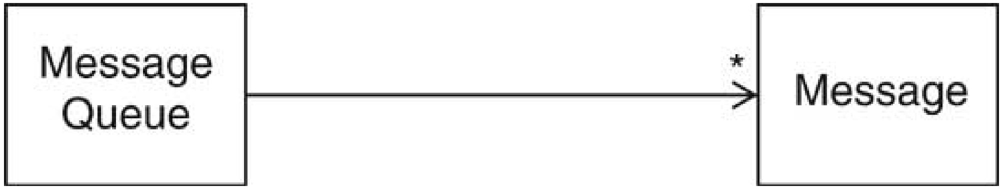
Association

- Some designers don't like aggregation
- More general association relationship
- Association can have roles

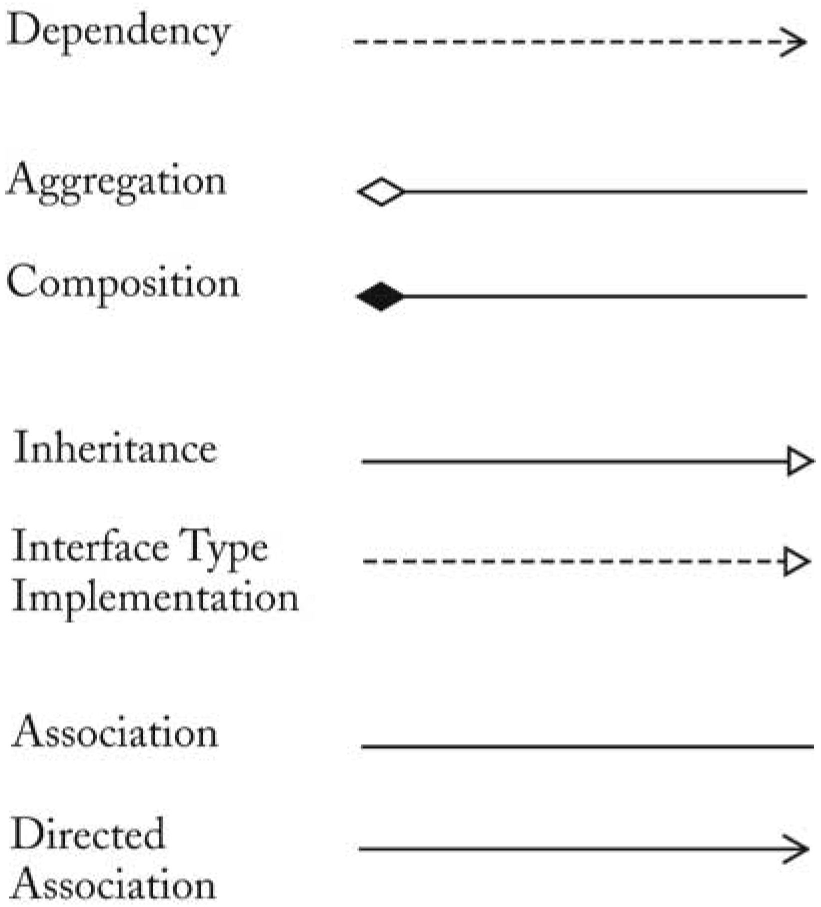


Association

- Some associations are bidirectional
  Can navigate from either class to the other
- Example: Course has set of students, student has set of courses
- Some associations are directed
  Navigation is unidirectional
- Example: Message doesn't know about message queue containing it



Class Relationships



Instance field

extends in java

implements in java

Object Relationships (1)

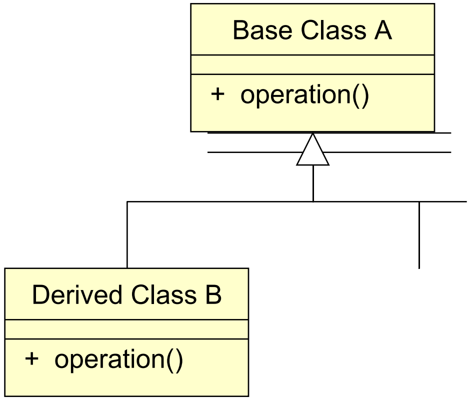
- Aggregation/Composition (HasA):

  Using instance variables that are references to other objects

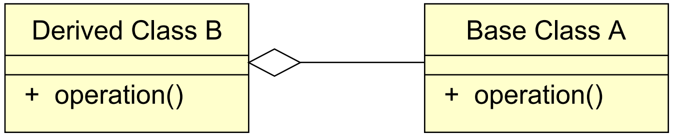
- Inheritance (IsA):

  Inheriting common properties through class extension

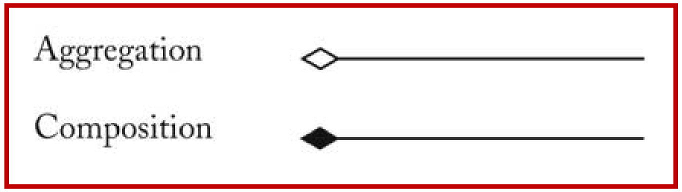
Inheritance:



Aggregation/Composition:



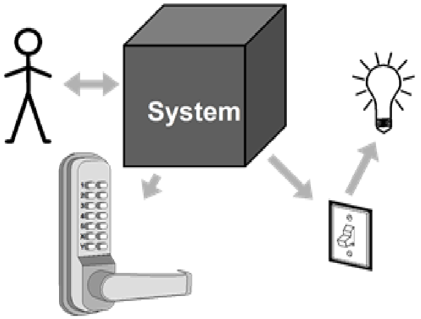
B acts as “front-end” for A and uses services of A (i.e., B may implement the same interface as A)

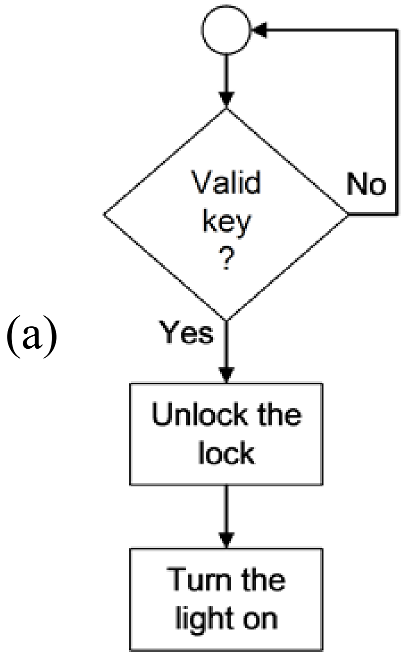


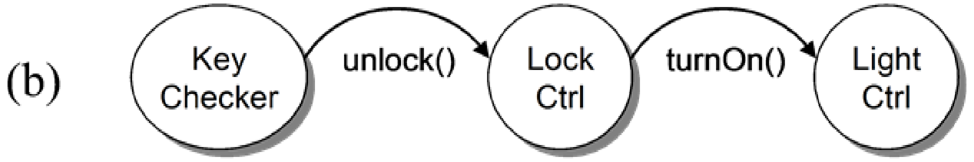
Object Relationships (2)

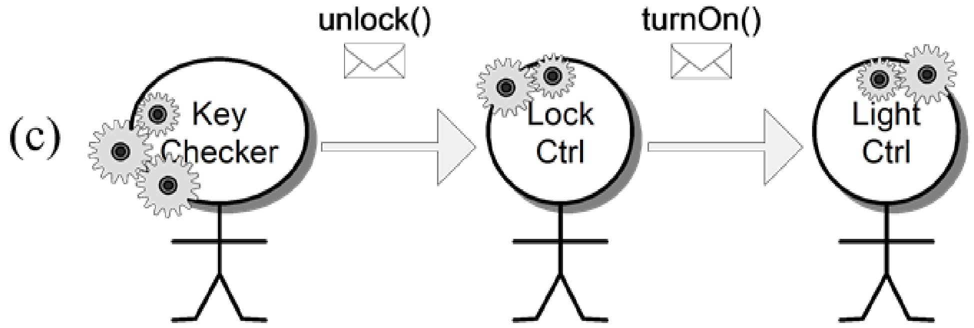
- Both inheritance and composition **extend** (traditional recognition out of date) the base functionality provided by another object
- INHERITANCE (IsA): Change in the “base” class propagates to the derived class and its client classes
  - BUT, any code change has a risk of unintentional introducing of bugs.
- AGGREGATION/COMPOSITION (HasA): More adaptive to change, because change in the “base” class is easily “contained” and hidden from the clients of the front-end class

Object-Oriented versus Process-Oriented Approaches









Process oriented

Object oriented

Object vs. Process-Oriented (1)

- **Process-oriented** is more intuitive because it is person-centric
  - thinking what to do next, which way to go
- **Object-oriented** may be more confusing because of labor-division
  - Thinking how to break-up the problem into tasks, assign responsibilities, and coordinate the work
  - It’s a management problem…

Object vs. Process-Oriented (2)

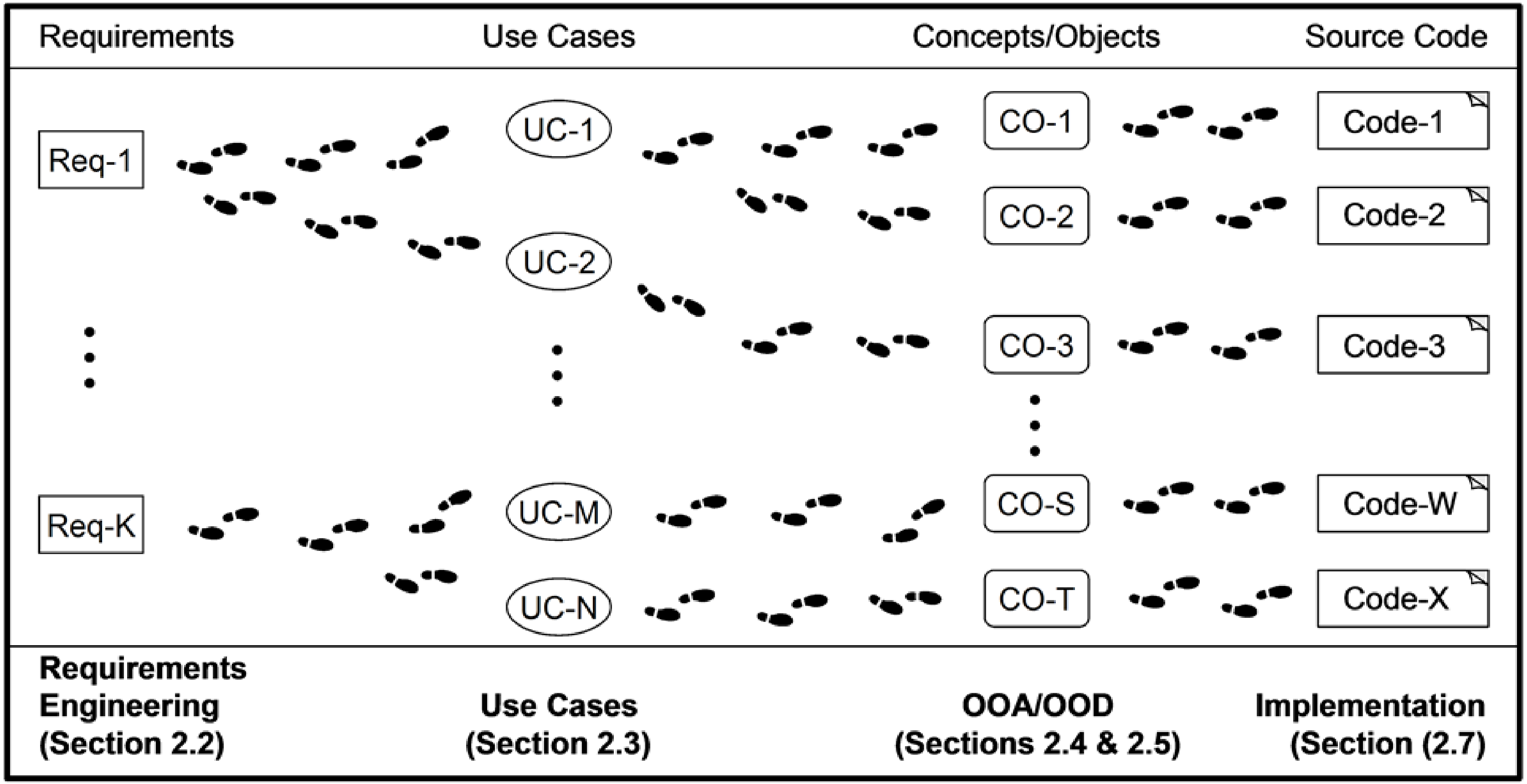
- **Process-oriented** does not scale to complex, large-size problems
  - Individual-centric, but…
- Large scale problems require organization of people instead of individuals working alone
- **Object-oriented** is organization-centric
  - But, hard to design well organizations…

How To Design Well OO Systems?

- That’s the key topic of this course!
- Decisive Methodological Factors:
  - Traceability
  - Testing
  - Measurement
  - Security

(Section 2.1.2)

Traceability (1)



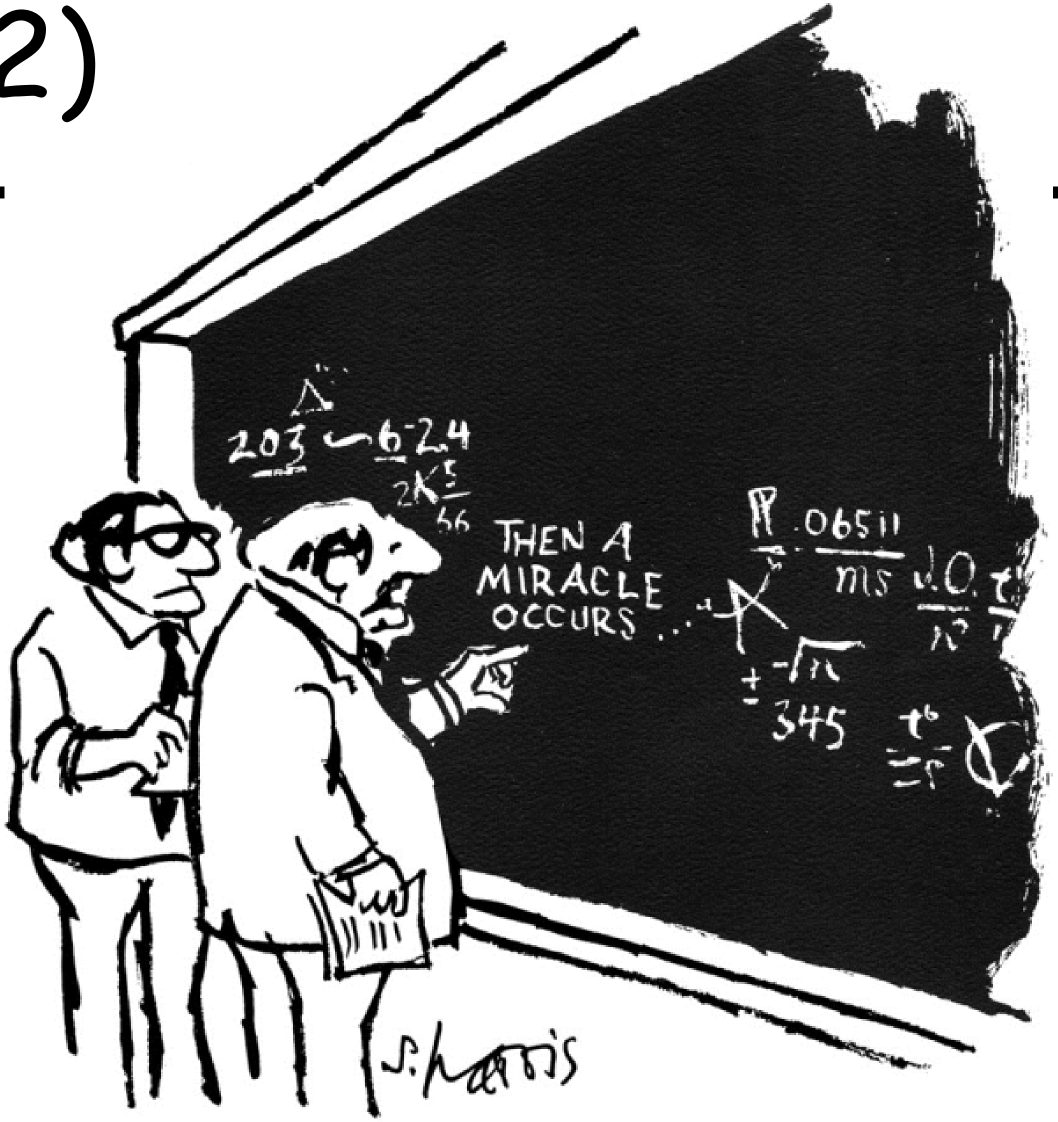
It should be possible to **trace** the evolution of the system, step-by-step, from individual requirements, through design objects, to code blocks.

Traceability (2)

Avoid inexplicable leaps!

...where did this come from?!

"Deus ex machina"



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

Testing (1)

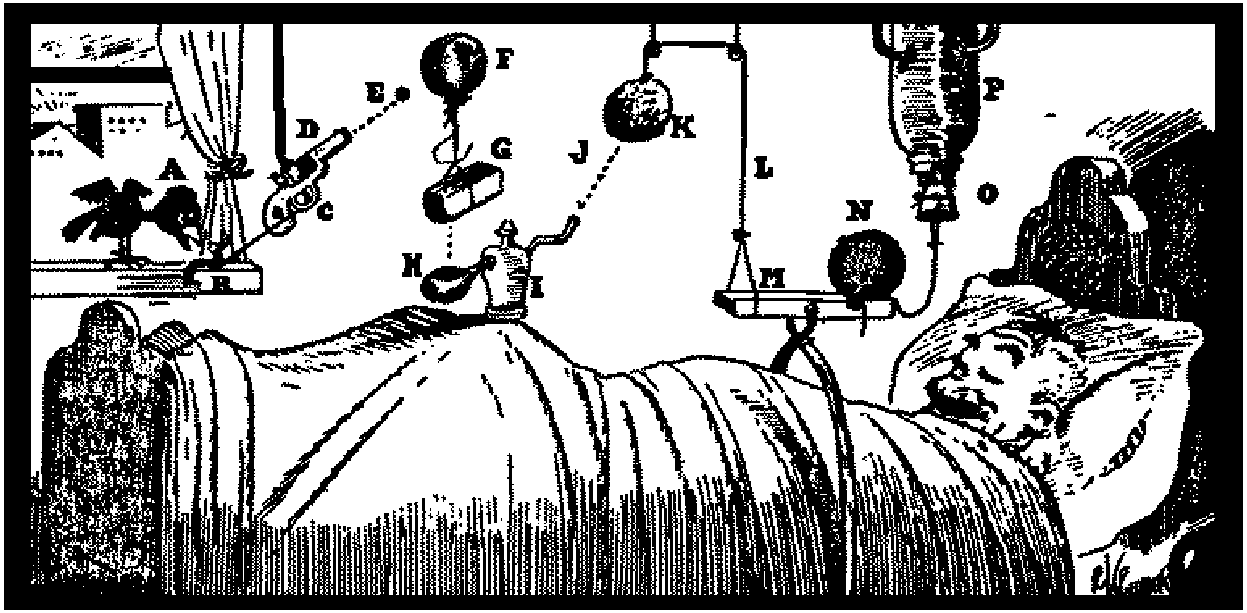
- **Test-Driven Development (TDD)**
- Every step in the development process must start with a plan of how to verify that the result meets a goal
- The developer should not create a software artifact (a system requirement, a UML diagram, or source code) unless they know how it will be tested

But, testing is not enough…

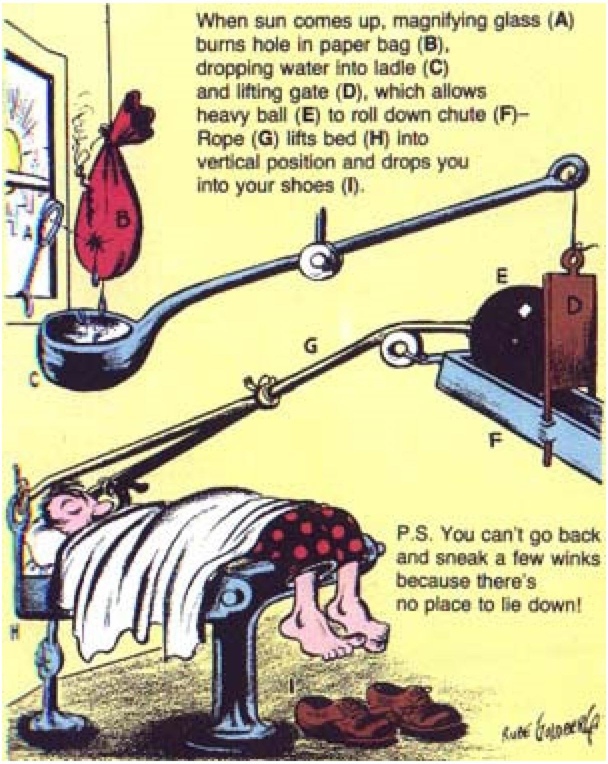
Testing (2)

A Rube Goldberg machine follows Test-Driven Development (TDD) —the *test case* is always described

…it’s fragile— works correctly for one scenario



Automatic alarm clock

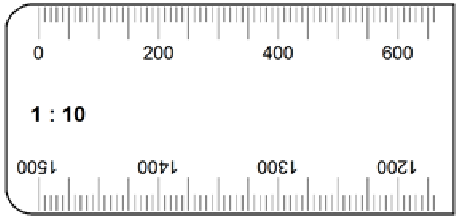

When sun comes up, magnifying glass (A) burns hole in paper bag (B), dropping water into ladle (C) and lifting gate (D), which allows heavy ball (E) to roll down chute (F)– Rope (G) lifts bed (H) into vertical position and drops you into your shoes (I).
P.S. You can't go back and sneak a few winks because there's no place to lie down!
A
B
C
D
E
F
G
H
I
Rube Goldberg

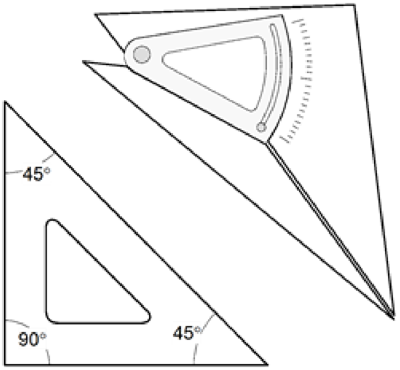
Oversleeping cure

Measuring (1)

- We need tools to monitor the **product quality**

- And tools to monitor the **developers productivity**

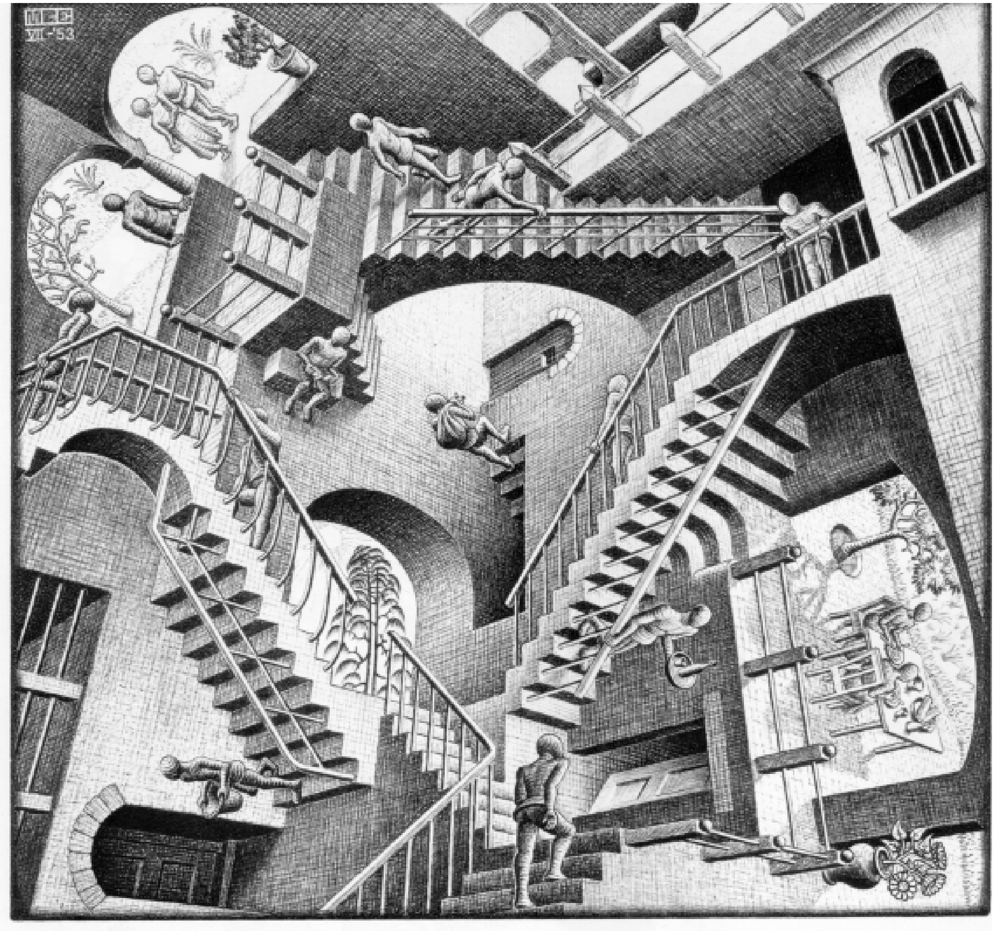




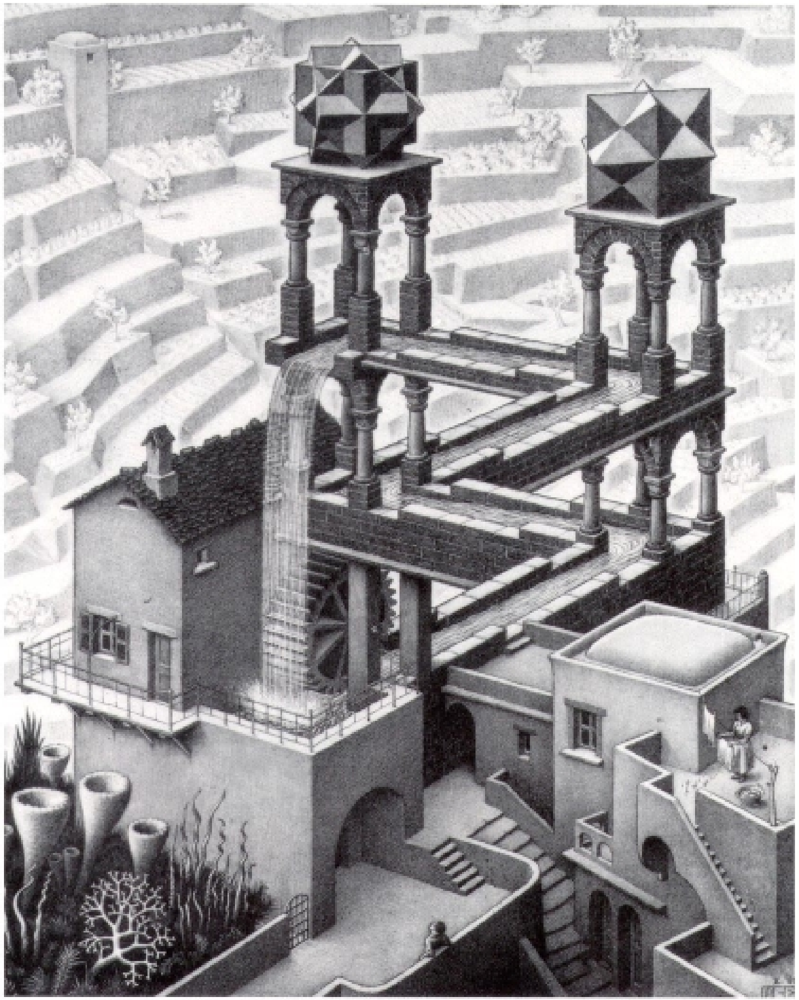
But, measuring is not enough…

Measuring (2)

Maurits Escher designs, work under all scenarios (incorrectly)
—robust but impossible



Relativity



Waterfall

Security

Conflicting needs
of computer security...

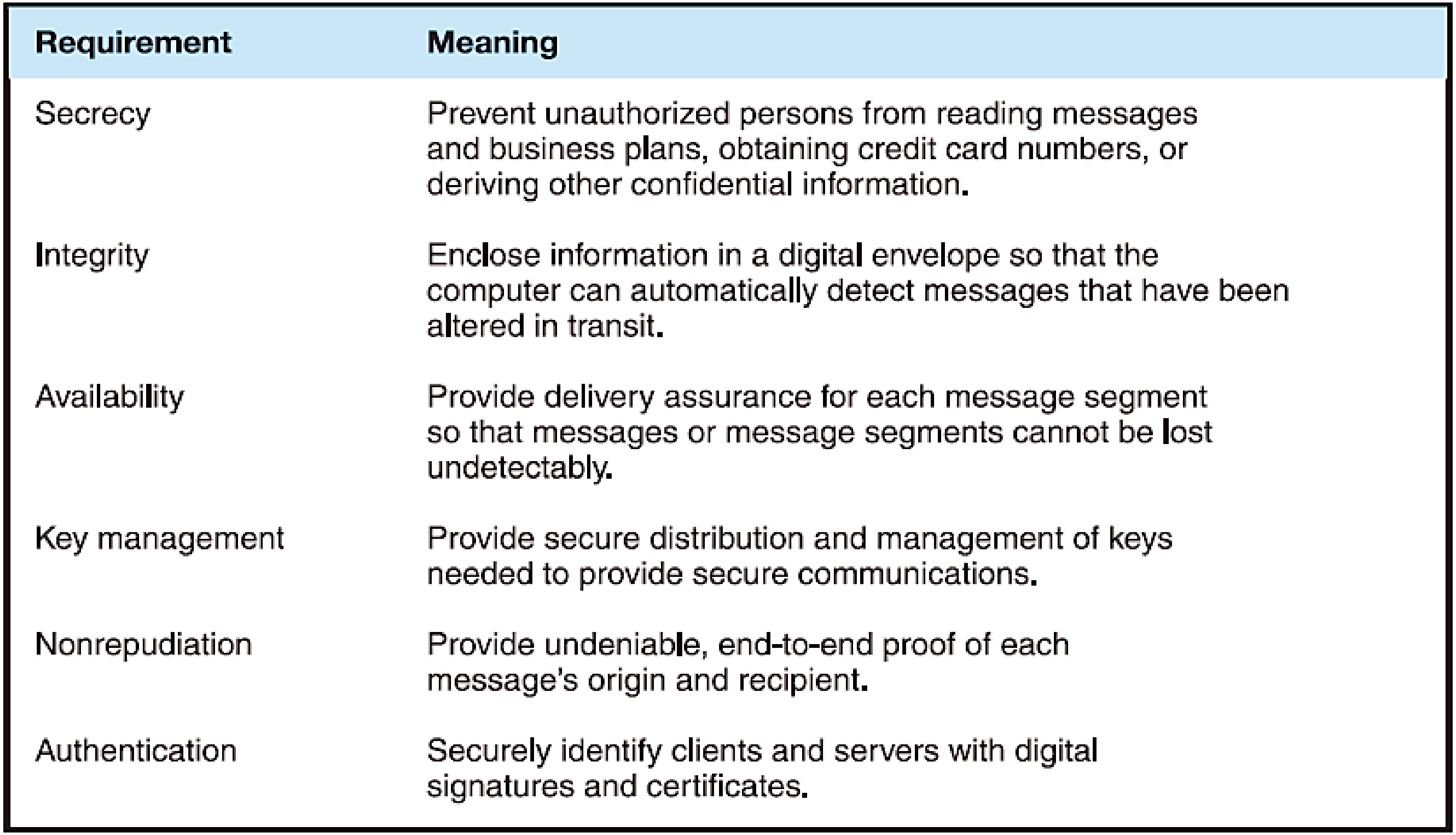
© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com
Loren Fishman
search-ID: lfin228
SECURITY PROTECTED
WELCOME

Microsoft Security Development Lifecycle (SDL)
http://www.microsoft.com/security/sdl/

Elements of Computer Security

- **Secrecy**
  - Protecting against unauthorized data disclosure
  - Ensuring data source authenticity
- **Integrity**
  - Preventing unauthorized data modification
  - **Man-in-the-middle exploit**
    - E-mail message intercepted; contents changed before forwarded to original destination
- **Necessity**
  - Preventing data delays or denials (removal)
  - Delaying message or completely destroying it

Requirements for secure electronic commerce



Requirement	Meaning
Secrecy	Prevent unauthorized persons from reading messages and business plans, obtaining credit card numbers, or deriving other confidential information.
Integrity	Enclose information in a digital envelope so that the computer can automatically detect messages that have been altered in transit.
Availability	Provide delivery assurance for each message segment so that messages or message segments cannot be lost undetectably.
Key management	Provide secure distribution and management of keys needed to provide secure communications.
Nonrepudiation	Provide undeniable, end-to-end proof of each message's origin and recipient.
Authentication	Securely identify clients and servers with digital signatures and certificates.

Security and Dependability of Sociotechnical Systems

- Dependability properties
  - The system attributes that lead to dependability.
- Availability and reliability
  - Systems should be available to deliver service and perform as expected.
- Safety
  - Systems should not behave in an unsafe way.
- Security
  - Systems should protect themselves and their data from external interference.