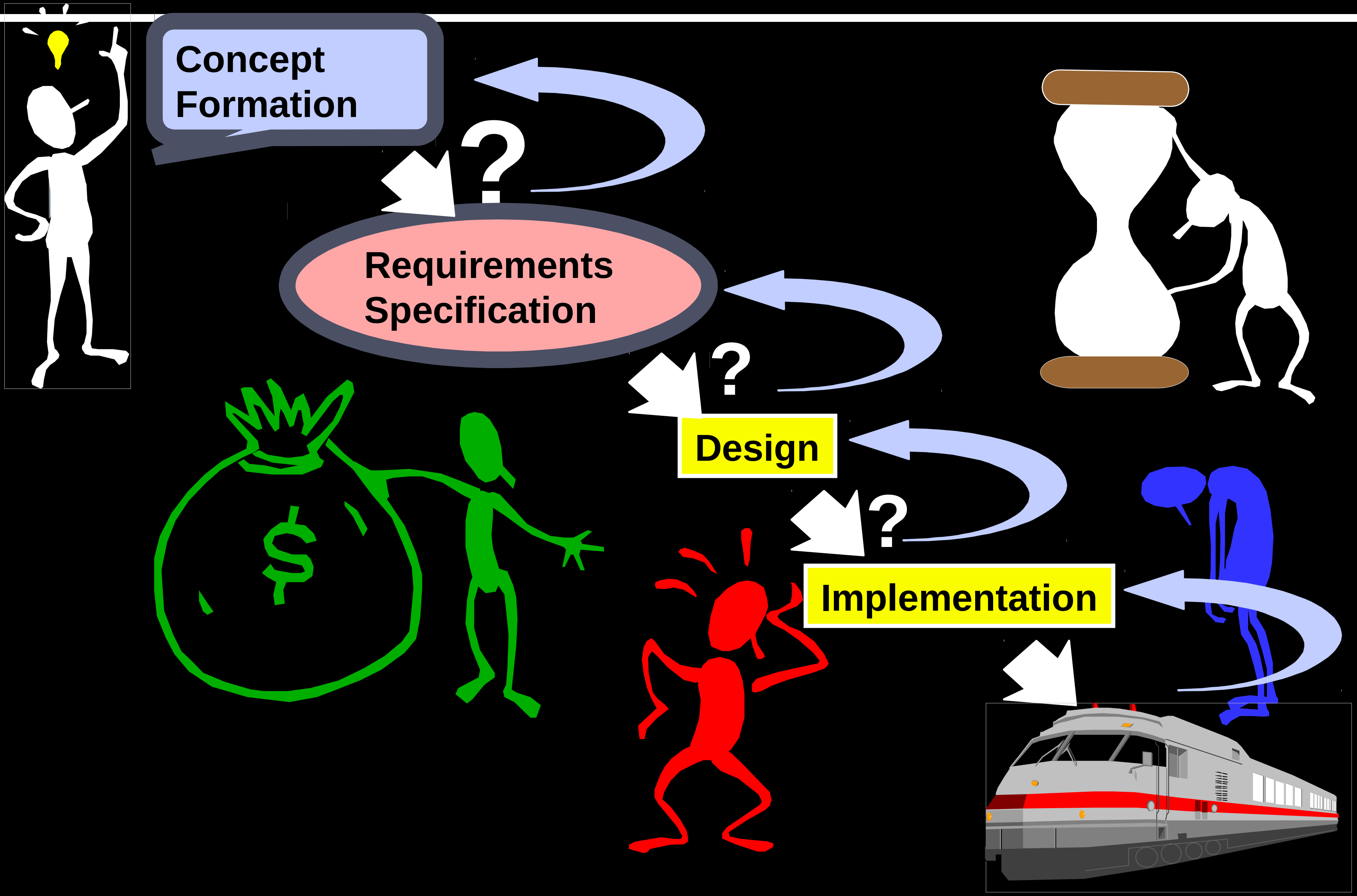
Ajitha Rajan

Inf2C-SE Summary Lecture

How Software Development Works



Why is Software Development so %$##% Hard? (L)

- Complexity
  - Software systems are the most complex artifacts ever created
- Invisibility
  - We cannot see the progress of the development
- Changeability
  - Software is “easy” to change
- Conformity
  - The software will have to be molded to fit whatever external constraints may be imposed

We Need a Software Process

- Structured set of activities required to develop a software system
  - Specification
  - Design
  - Validation
  - Evolution
- Activities vary depending on the organization and the type of system being developed
- Must be explicitly modeled if it is to be managed

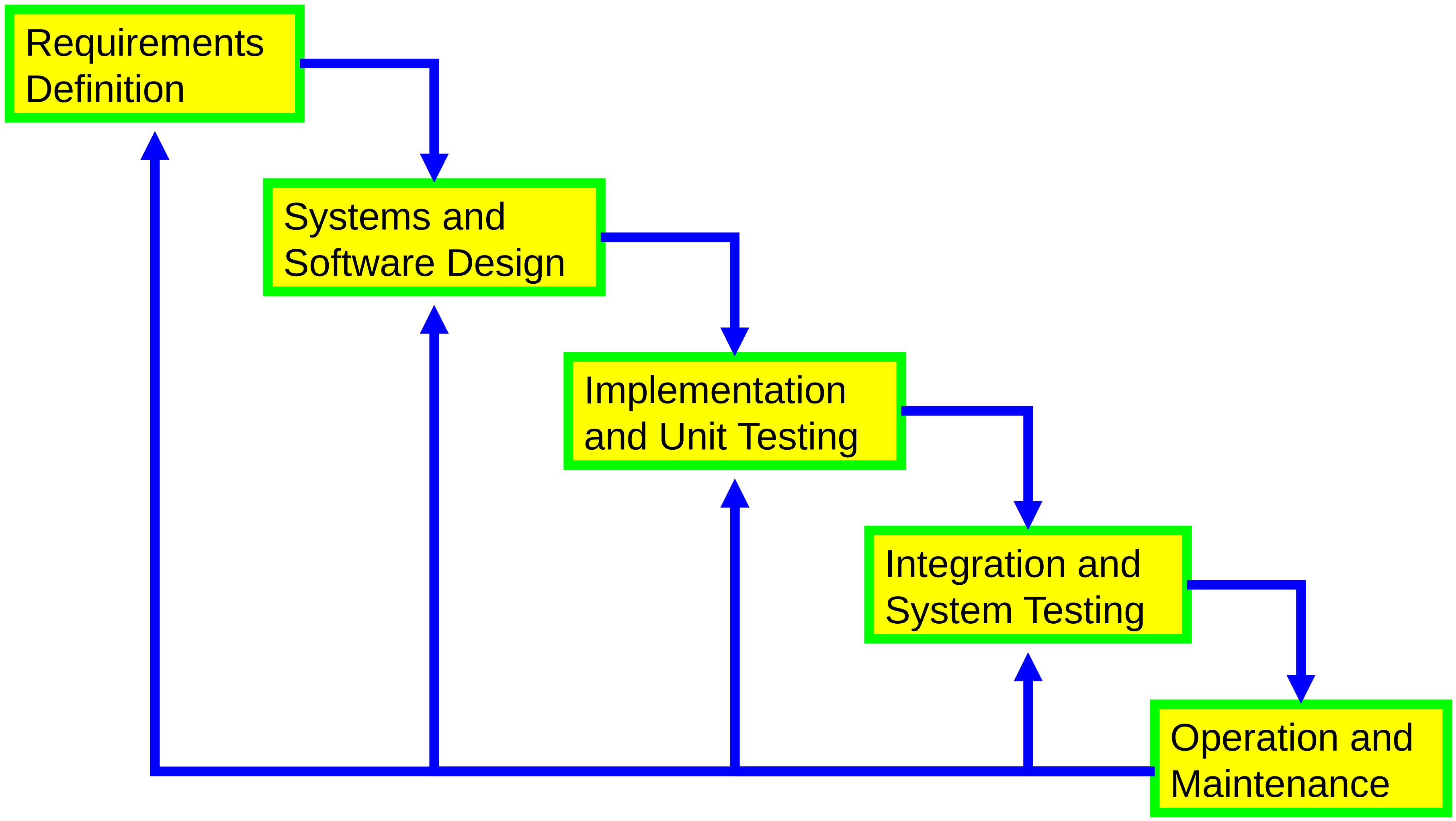
Generic Software Process Models

- The Waterfall Model
  - Separate and distinct phases of specification and development
- Evolutionary Development
  - Specification and development are interleaved
- Spiral Model
  - Let risk analysis drive your process
- Incremental Development
  - Deliver your system in small planned increments
- Agile and eXtreme Programming

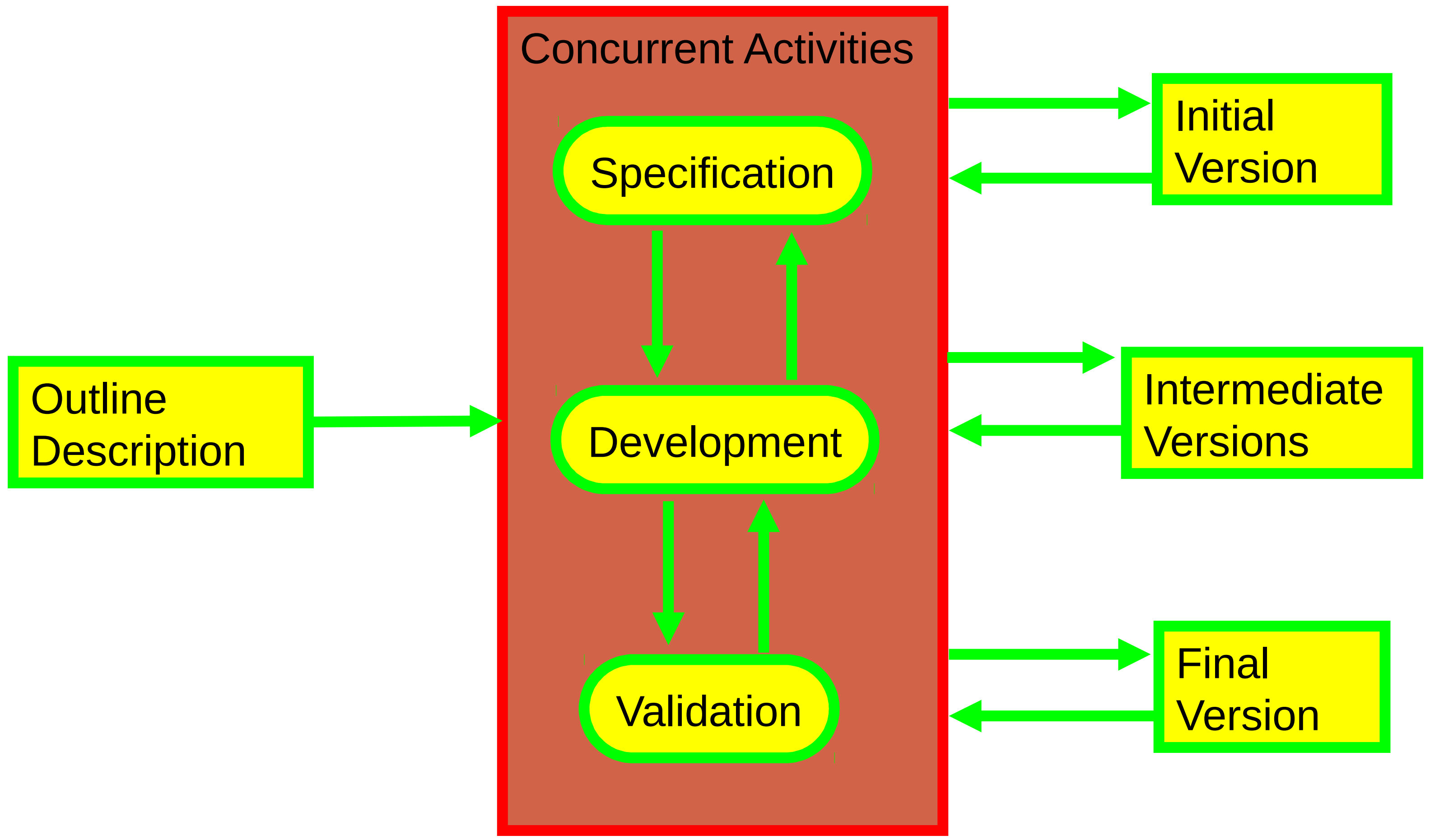
Process Characteristics

- Understandability
  - Is the process defined and understandability
- Visibility
  - Is the process progress externally visible
- Supportability
  - Can the process be supported by CASE tools
- Acceptability
  - Is the process acceptable to those involved in it
- Reliability
  - Are process errors discovered before they result in product errors
- Robustness
  - Can the process continue in spite of unexpected problems
- Maintainability
  - Can the process evolve to meet changing organizational needs
- Rapidity
  - How fast can the system be produced

Waterfall Model


Requirements Definition
Systems and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

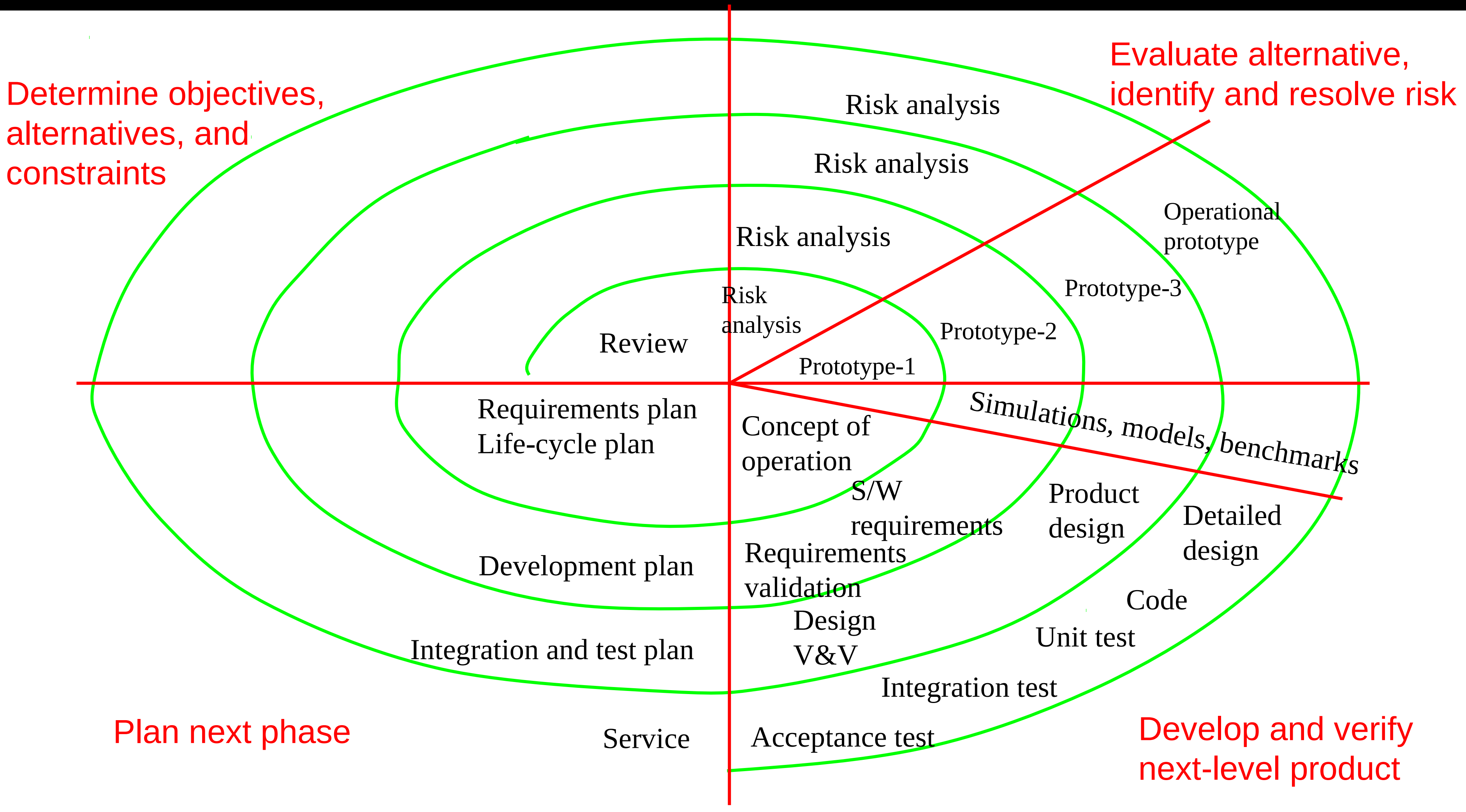
Evolutionary Development



Evolutionary Development

- Evolutionary prototyping
  - Objective is to work with customers and to evolve a final system from an initial outline specification.
  - Typically starts with well-understood requirements
- Throw-away prototyping
  - Objective is to understand the system requirements.
  - Typically starts with poorly understood requirements

Spiral Model

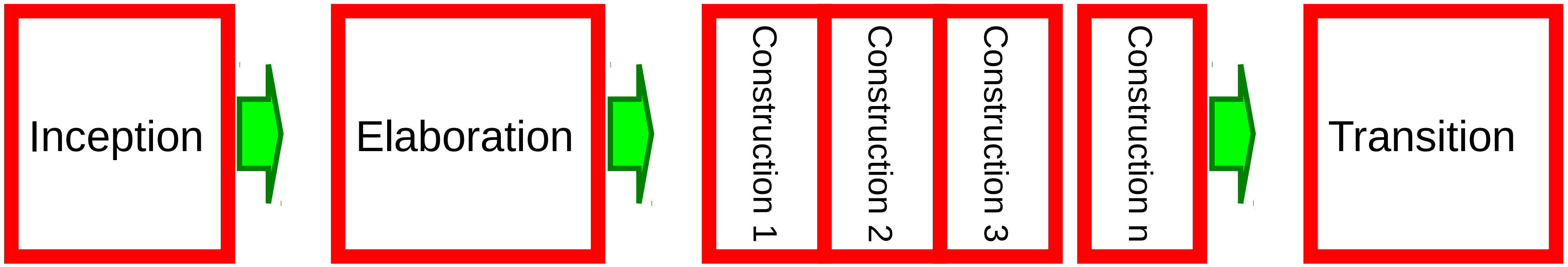


Incremental Development

- System is developed and delivered in increments after establishing an overall architecture
- Users may experiment with delivered increments while others are being developed
  - Therefore, these serve as a form of prototype system
- Intended to combine some of the advantages of prototyping but with a more manageable process and better system structure

Process Overview

- Inception
- Elaboration
- Construction
    - Many iterations
- Transition



Agile processes

What the spiral models were reaching towards was that software development has to be *agile*: able to react quickly to change.

The Agile Manifesto `http://agilemanifesto.org`:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
> **Individuals and interactions** *over processes and tools*
> **Working software** *over comprehensive documentation*
> **Customer collaboration** *over contract negotiation*
> **Responding to change** *over following a plan*
> *That is, while there is value in the items on the right, we value the items on the left more.*

12 principles of Agile

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily co-operation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity- The art of maximizing the amount of work not done - is essential
- Self-organizing teams
- Regular adaptation to changing circumstances

Extreme Programming

One variant: Extreme Programming (XP) is

“a humanistic discipline of software development, based on values of communication, simplicity, feedback and courage”

People: Kent Beck, Ward Cunningham, Ron Jeffries, Martin Fowler, Erich Gamma...

More info: `www.extremeprogramming.org`,
Beck “Extreme Programming Explained: Embrace Change”

XP Practices

The Planning Game
Small releases
Metaphor
Simple design
Testing
Refactoring
Pair programming
Collective ownership
Continuous integration
40-hour week
On-site customer
Coding standards

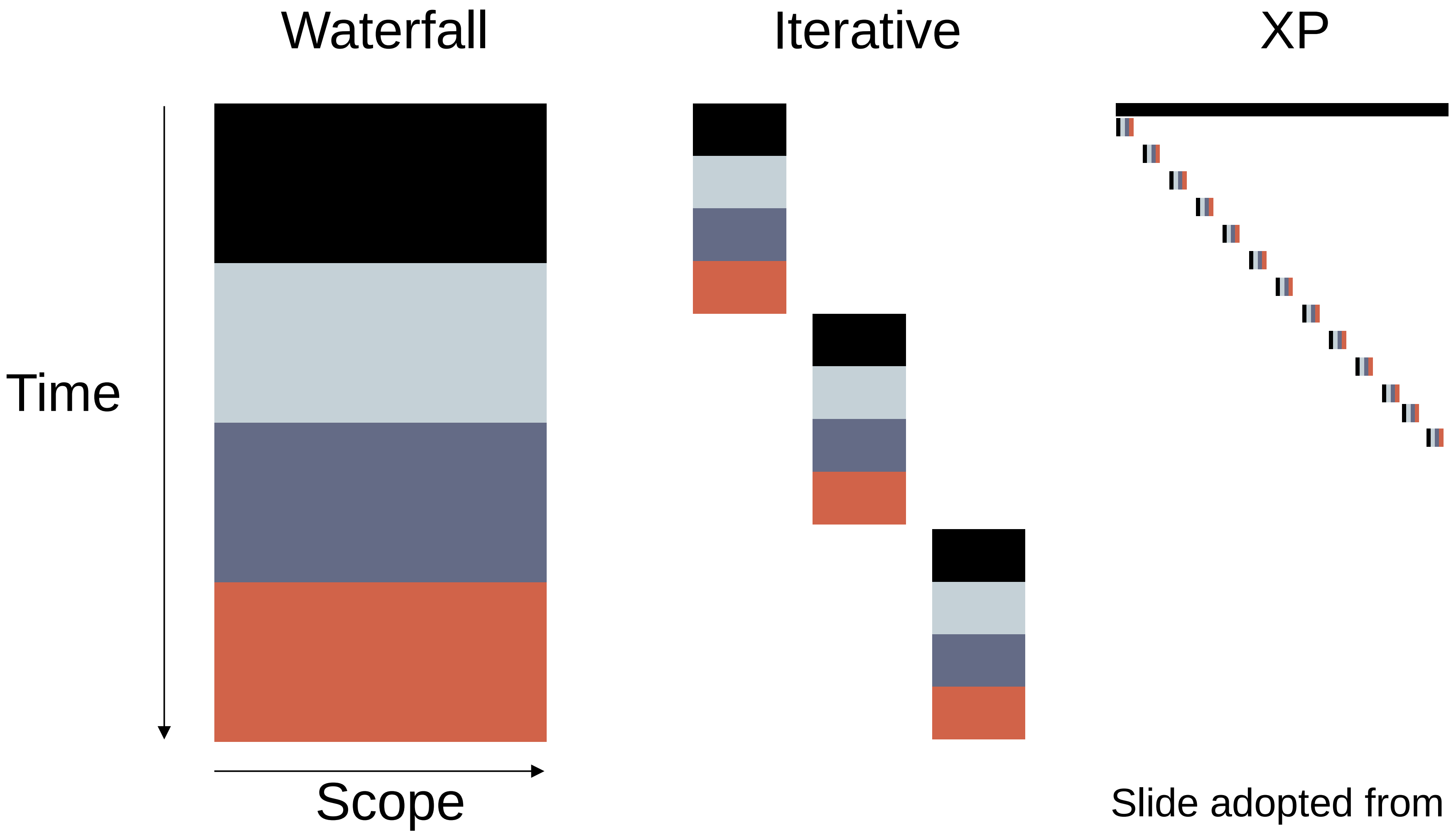
Where is XP applicable?

The scope of situations in which XP is appropriate is somewhat controversial. Two examples

- there are documentated cases where it has worked well for development in-house of custom software for a given organisation (e.g. Chrysler).
- A decade ago it seemed clear that it wouldn't work for Microsoft: big releases were an essential part of the business; even the frequency of updates they did used to annoy people. Now we have automated updates to OSs, and Microsoft is a Gold Sponsor of an Agile conference

XP does need: team in one place, customer on site, etc. "Agile" is broader.

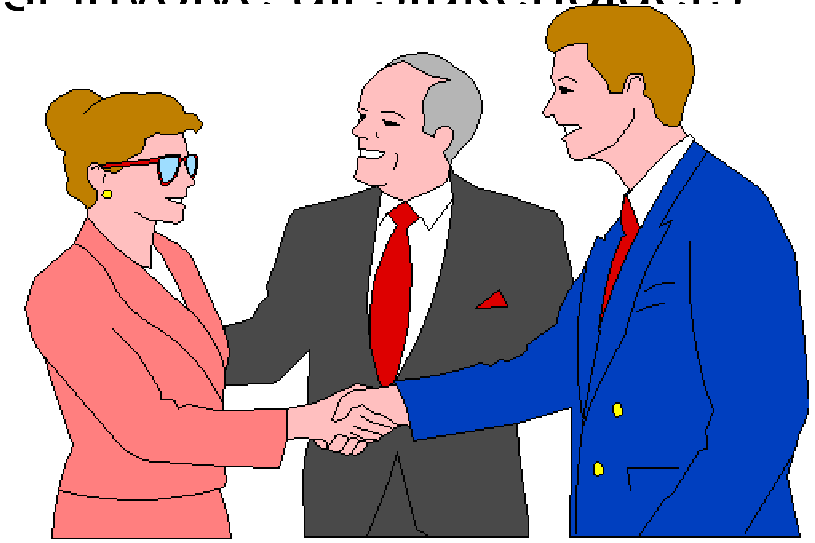
Three Processes



Slide adopted from Beck

Requirements Specification

- High-level description of what a system should do
- Must be detailed enough to distinguish between the “right” and the “wrong” system
- Capture the **what** not the **how**
- The specification process must involve all stakeholders
  - Customers
  - Engineers
  - Regulatory agencies
  - Users



Key Points

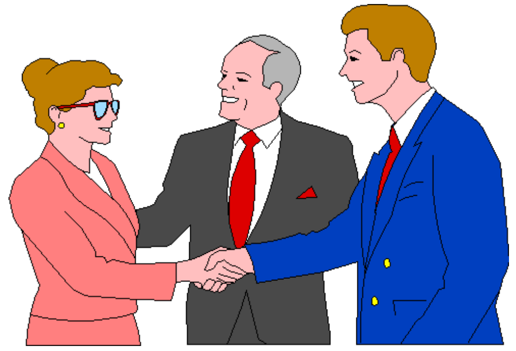
- Requirements capture what a proposed system shall do
  - But avoids design detail as much as possible
  - Written in the user's language

- **Poor requirements are the source of all evil**

- Requirements problems are the
  - Most costly
  - Most difficult to correct (they are conceptual)

Requirements Readers

Market Requirements Definition	Client managers System end-users Client engineers Contractor managers System architects
Software Requirements Specification	System end-users Client engineers System architects Software developers
Software Design Description	Client engineers System architects Software developers



Capturing Good Requirements

3 Common Problems

- Poorly structured requirements document
- Poorly written individual requirements
- Untestable requirements (future lecture)

Four Easy Requirements Guidelines

- Avoid requirements “fusion”
  - One requirement per requirement specification
- Be precise
  - No vague requirements
- Be rigorous in defining requirements test cases
  - If you cannot define how to test if a requirement is satisfied, you probably have a poor requirement
- Attach a person to each requirement
  - People are much less likely to add “the kitchen sink” if their name is there – no gold plating

Each Requirement Must Be

- Correct
  - The requirement is free from faults.
- Precise, unambiguous, and clear
  - Each item is exact and not vague; there is a single interpretation; the meaning of each item is understood; the specification is easy to read.
- Complete
  - The requirement covers all aspects of the user function.
- Consistent
  - No item conflicts with another item in the specification.

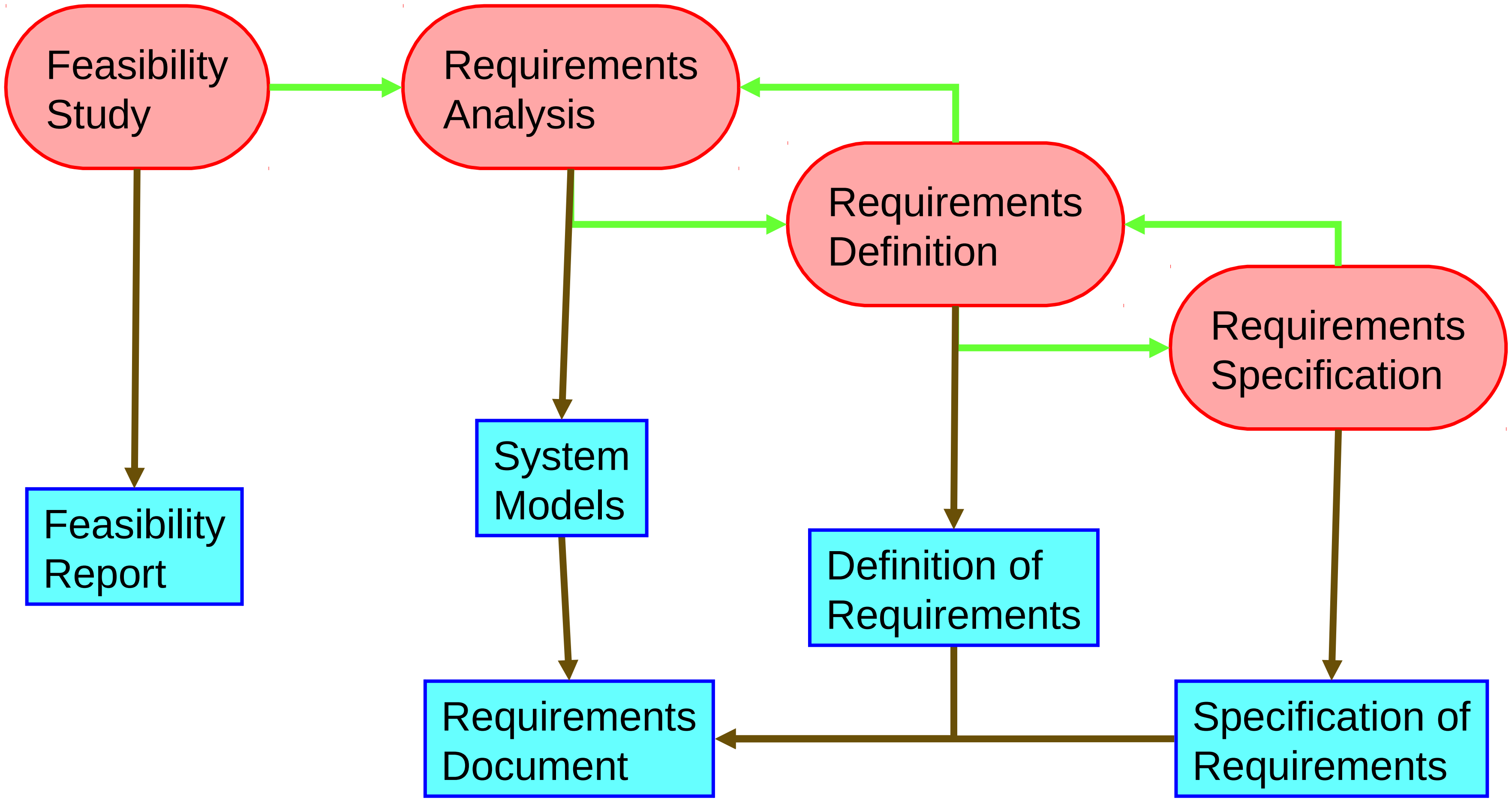
Each Requirement Must Be (Cont.)

- Relevant
  - Each item is pertinent to the problem and its solution.
- Testable
  - During program development and acceptance testing, it will be possible to determine whether the item has been satisfied.
- Traceable
  - Each item can be traced to its origin in the problem environment.
- Feasible
  - Each item can be implemented with the available techniques, tools, resources, and personnel, and within the specified cost and schedule constraints

The SRS (as a document) Must Be

- Complete
  - All user requirements have been included. Do not forget abnormal and boundary cases.
- Consistent
  - No item conflicts with another item in the specification.
- The requirements shall be at a consistent level of detail
- Manageable and Modifiable
  - Things will change and we must be able to accommodate the inevitable requirements evolution.

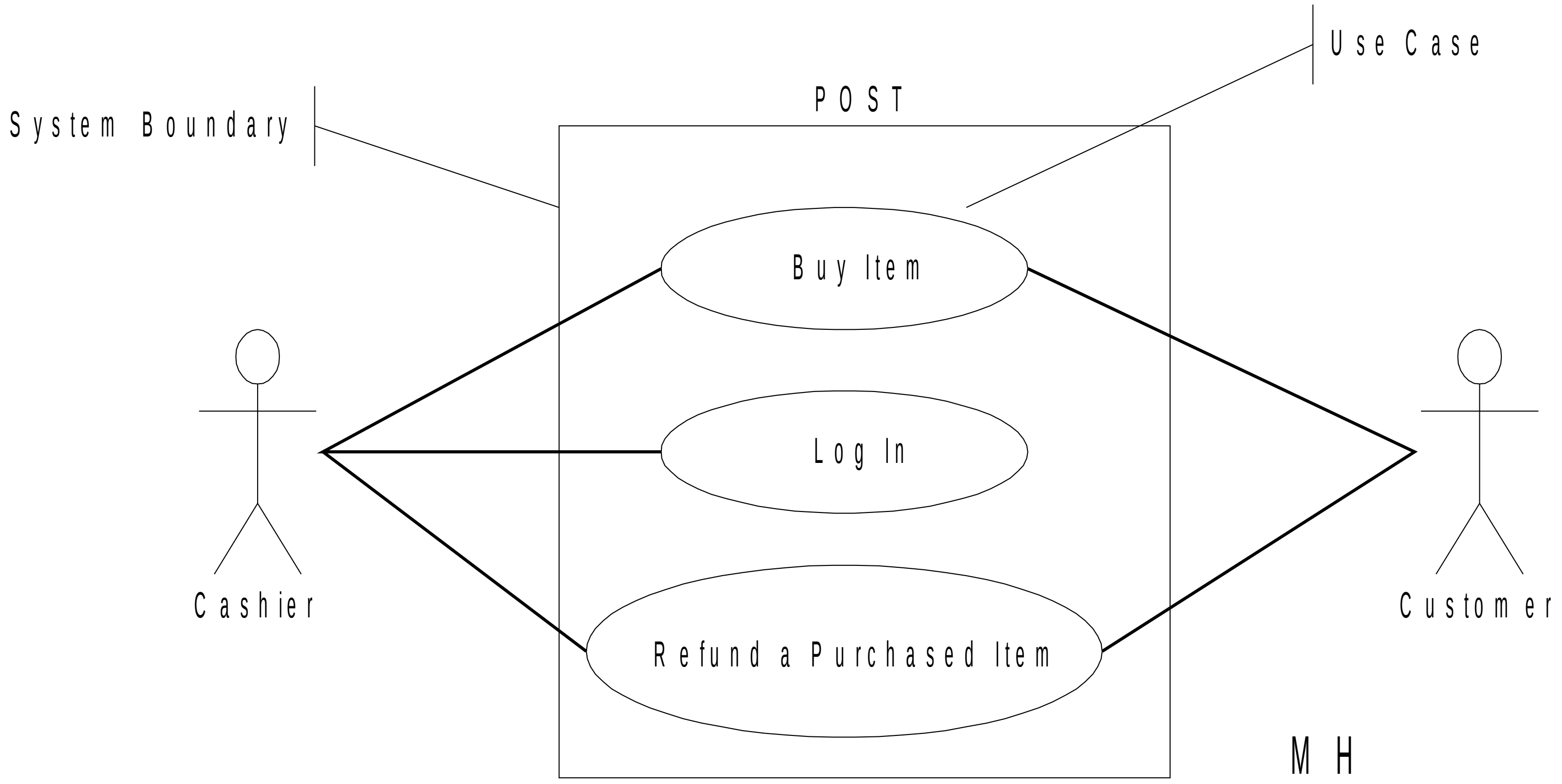
The Requirements Engineering Process



What is a Use-Case

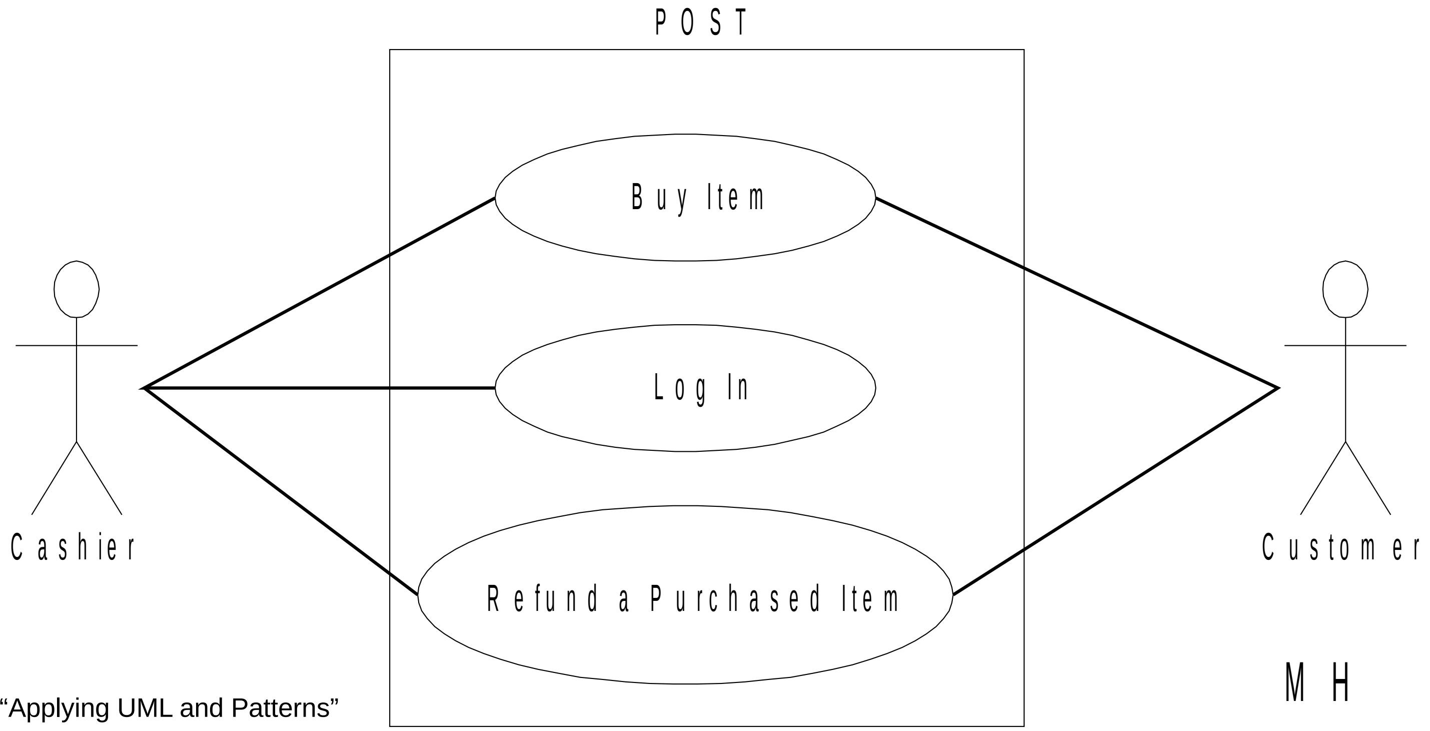
- A use-case captures some user visible function
- This may be a large or small function
  - Depends on the level of detail in your modeling effort
- A use-case achieves a discrete goal for the user
- Examples
  - Format a document
  - Request an elevator
- How are the use cases found (captured or elicited)?

Use-Case Diagrams



Setting the System Boundary

- The system boundary will affect your actors and use-cases



Adapted from Larman “Applying UML and Patterns”

Home Heating Scenario

Use case:	**Power Up**
Actors:	Home Owner (initiator)
Type:	Primary and essential
Description:	The Home Owner turns the power on. Each room is temperature checked. If a room is below the the desired temperature the valve for the room is opened, the water pump started, the fuel valve opened, and the burner ignited. If the temperature in all rooms is above the desired temperature, no actions are taken.
Cross Ref.:	Requirements XX, YY, and ZZ
Use-Cases:	None

When to use Use-Cases

- In short, always!!!
- Requirements is the toughest part of software development
  - Use-Cases is a powerful tool to understand
  - Who your users are (including interacting systems)
  - What functions the system shall provide
  - How these functions work at a high level
- Spend adequate time on requirements and in the elaboration phase

If you can’t test it, it is not a requirement!

Derive Test Cases for the Requirements

Test the Requirement

Test Case 1

Input

Artificially raise the temperature above *threshold*

Test procedure

Measure the time it takes for the *alarm* to come on

Expected output

The *alarm* shall be on within 2 seconds

Key Points

- Do yourself and the testing group a favor—**Develop Test Cases for Each Requirement**

- If the requirement cannot be tested, you most likely have a bad requirement
  - Rewrite so it is testable
  - Remove the requirement
  - Point out why this is an untestable requirement

- **Your requirements <u>and</u> testing effort will be greatly improved**

Mainly “Will It Work?”

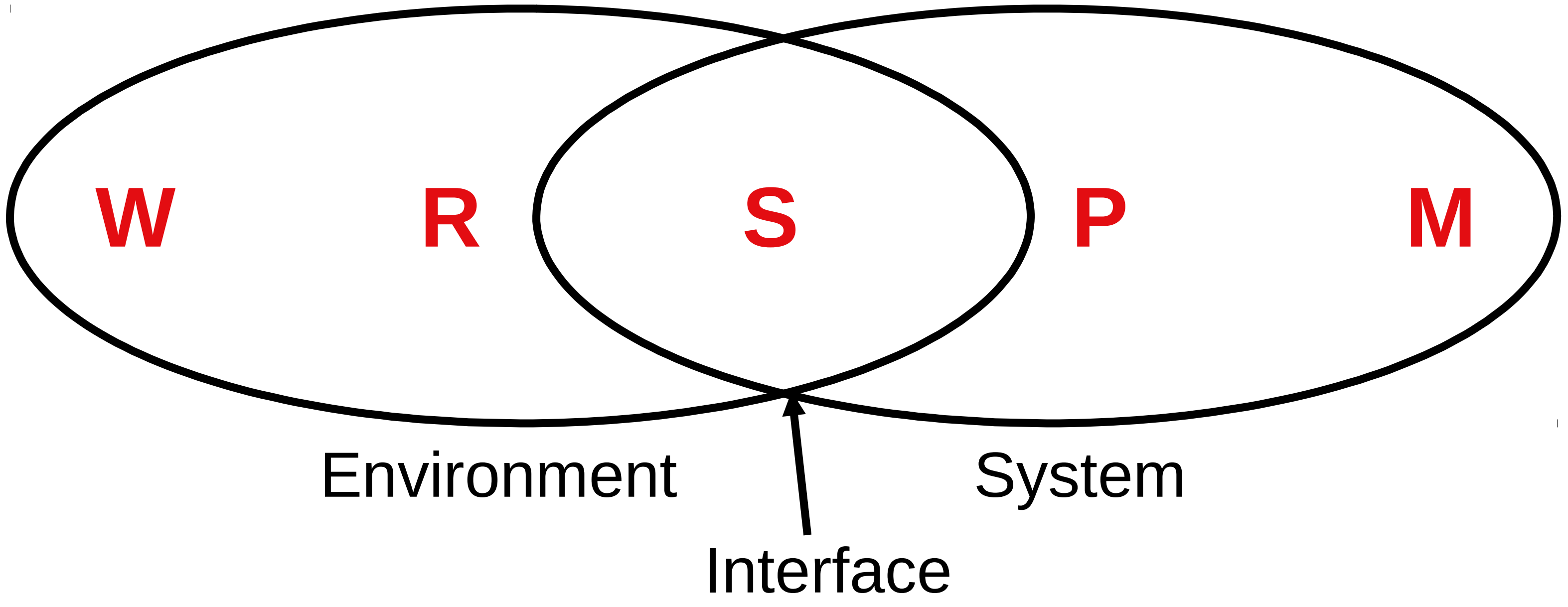
The World Machine Model

Capture the Right Thing

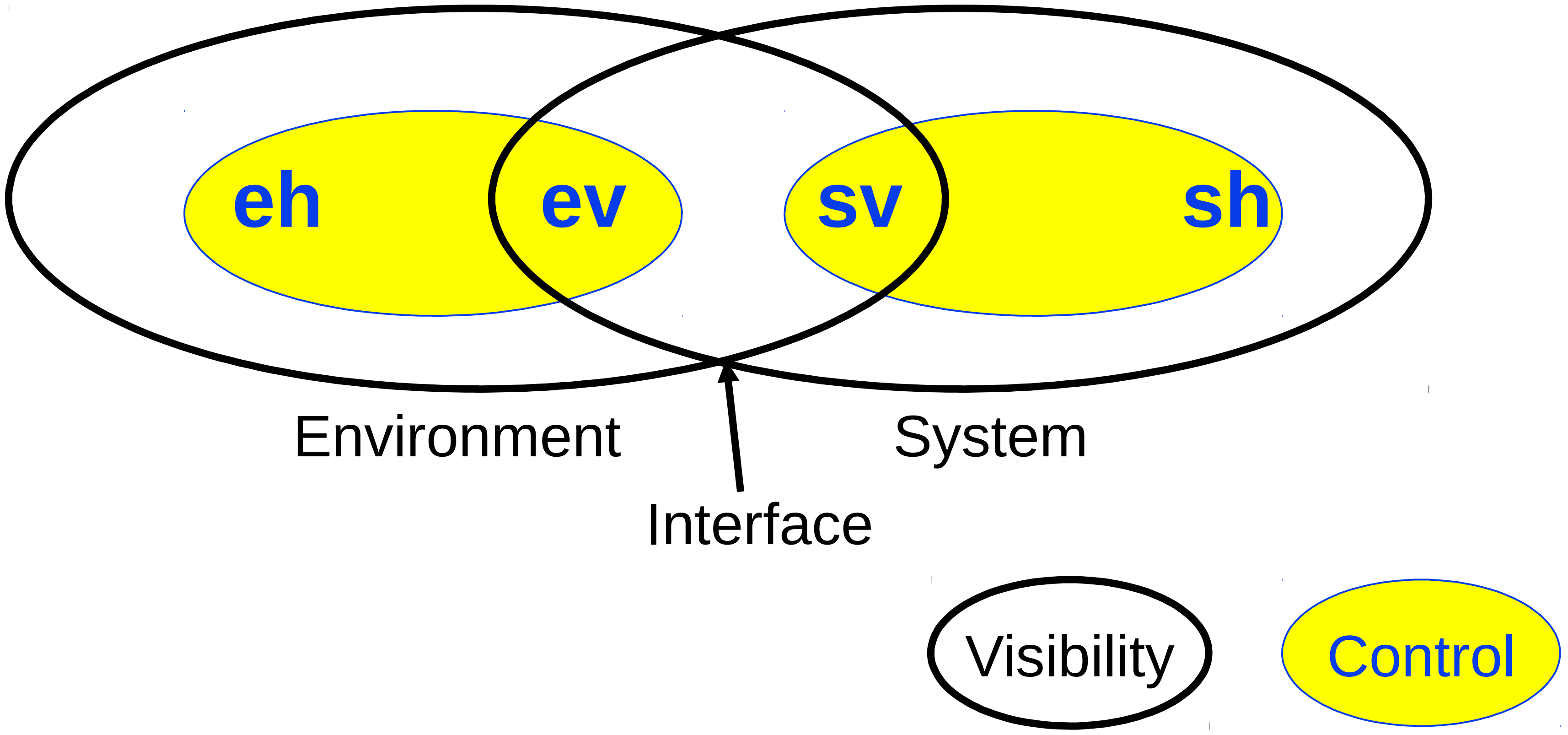
- Requirements are always in the system domain
- Software specification is in the computer domain
- There are several levels of abstraction in between
  - Abstract away some details but not others

The WRSPM Model

W – The World Assumptions (domain model)
R – The Requirements
S – The system specification
P – The Program (running on the machine)
M – The machine physically implementing the system



The Variables in WRSPM



Design Strategies

- Functional design
  - The system is designed from a functional viewpoint
  - The system state is centralized and shared between the functions operating on that state
- Object-oriented design
  - The system is viewed as a collection of interacting objects
  - The system state is de-centralized and each object manages its own state
  - Objects may be instances of an object class and communicate by exchanging messages

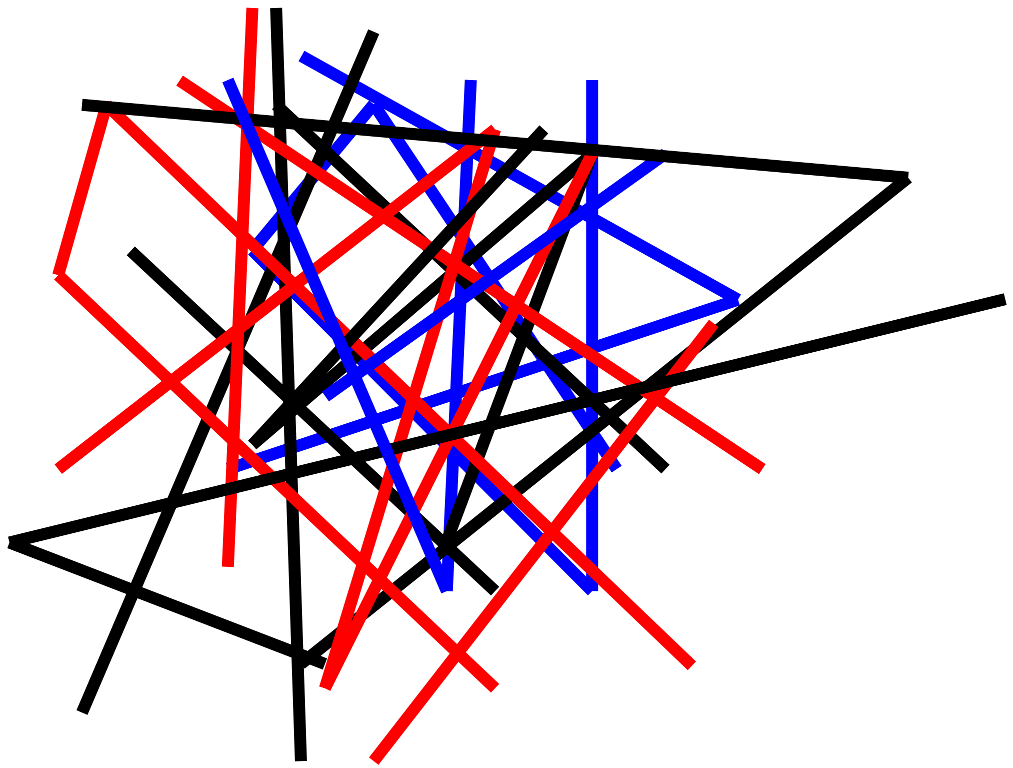
Key Points

- Design is a creative process
- Design activities include architectural design, system specification, component design, data structure design and algorithm design
- Functional decomposition considers the system as a set of functional units
- Object-oriented decomposition considers the system as a set of objects

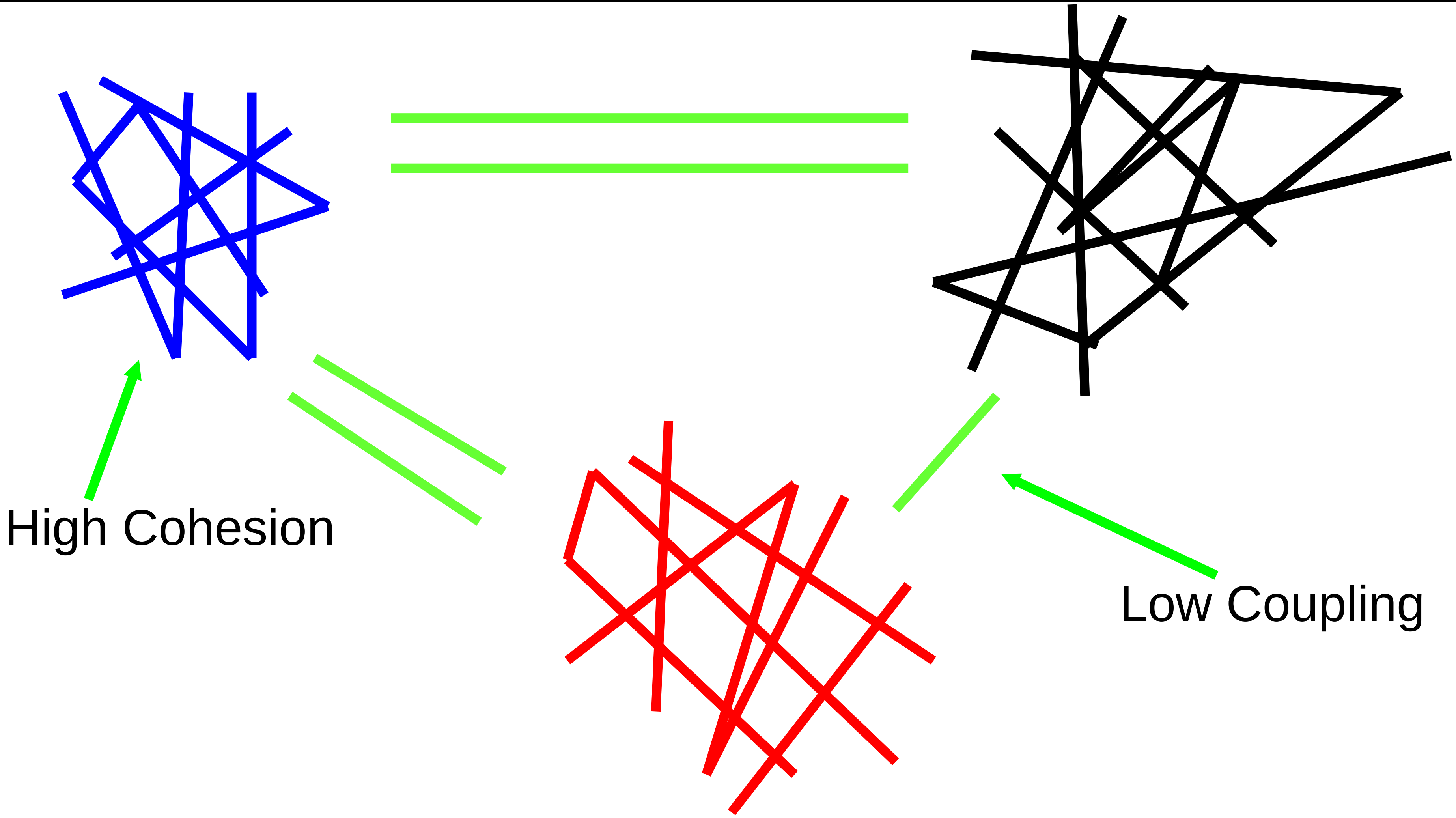
Objectives

- To discuss some design quality attributes
  - “Clarity”
  - Simplicity
  - Modularity
  - Coupling
  - Cohesion
  - Information hiding
  - Data encapsulation
  - “Ilities”
    - Adaptability
    - Traceability

More Modularity



Two Essential Properties

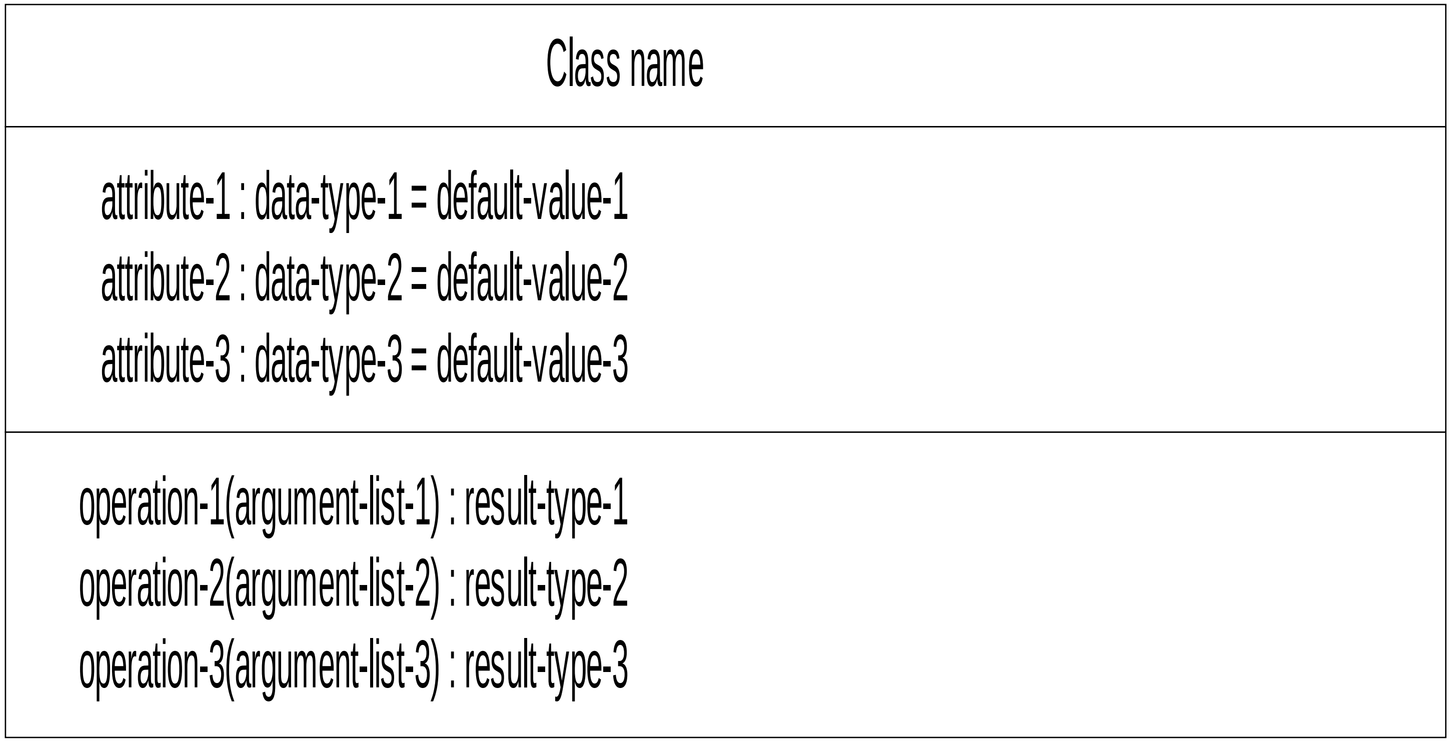


Several Complementary Models

- Structural Models
  - Describes the structure of the objects in a system
  - Structure of individual objects (attributes and operations)
  - Relationships between the objects (inheritance, sharing, and associations)
  - Clustering of objects in logical packages and on the actual hardware
- Dynamic models (behavioral models)
  - The aspects related to sequencing of operations
  - Changes to attributes and sequences of changes
  - The control aspects of the system

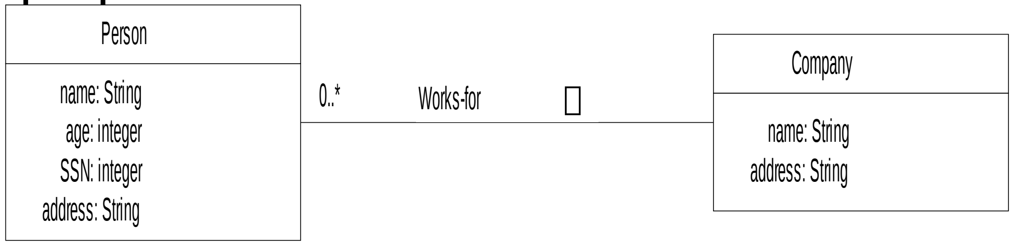
The Class Diagrams

Object Notation - Summary

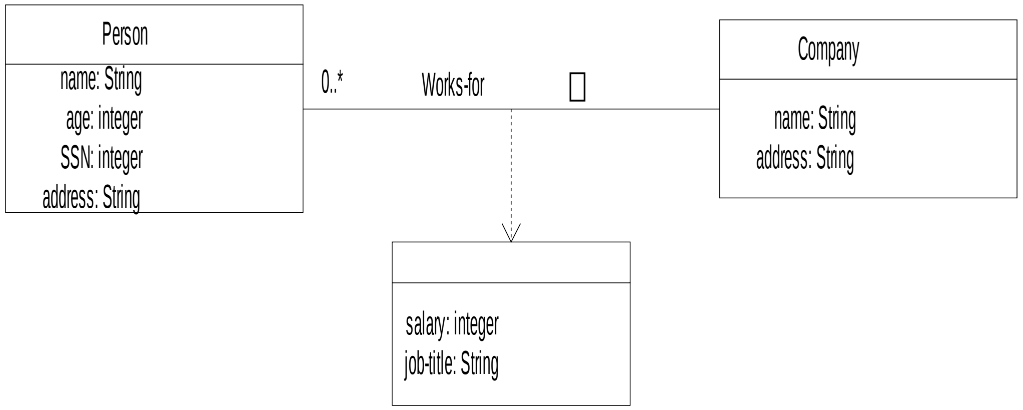

Class name
attribute-1 : data-type-1 = default-value-1
attribute-2 : data-type-2 = default-value-2
attribute-3 : data-type-3 = default-value-3
operation-1(argument-list-1) : result-type-1
operation-2(argument-list-2) : result-type-2
operation-3(argument-list-3) : result-type-3

Link Attributes

- Associations can have properties the same way objects have properties



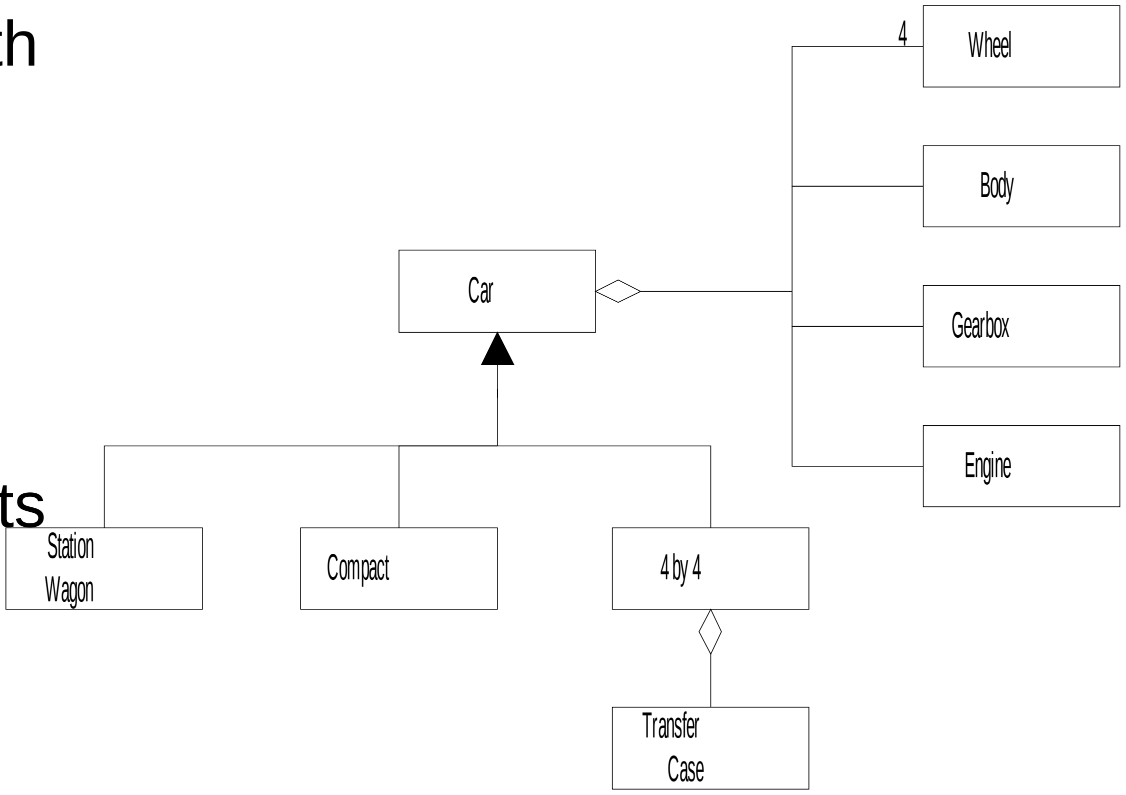
How represent salary and job title?



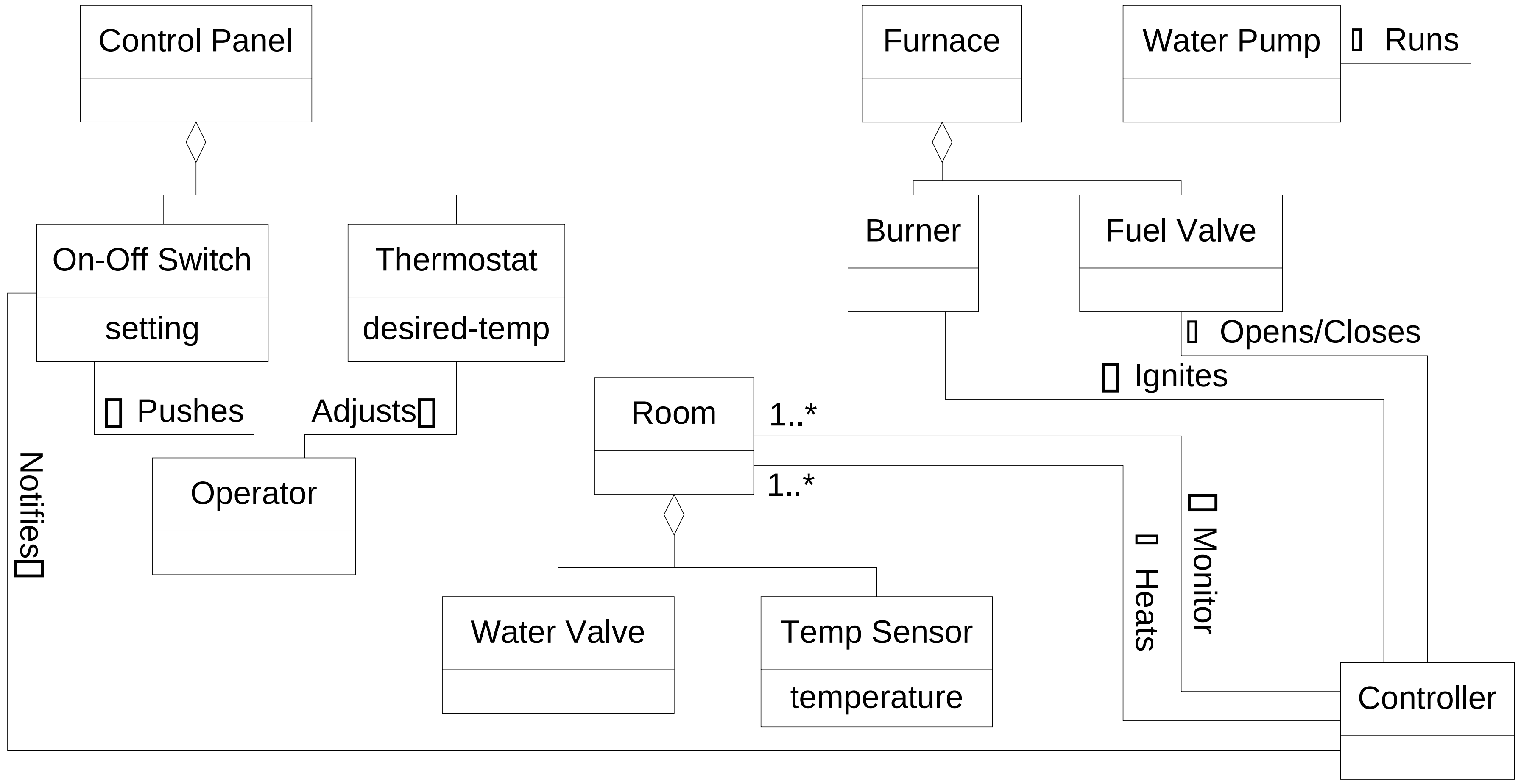
Use a link attribute!

Aggregation Versus Inheritance

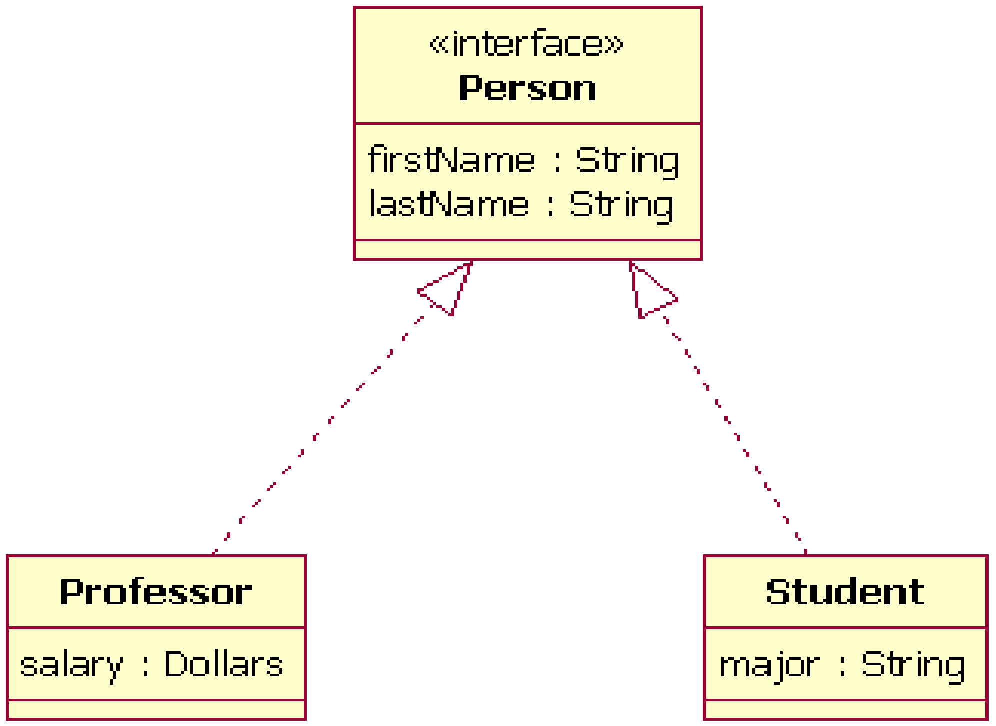
- Do not confuse the is-a relation (inheritance) with the is-part-of relation (aggregation)
- Use inheritance for special cases of a general concept
- Use aggregation for parts explosion



Class Diagram—v1



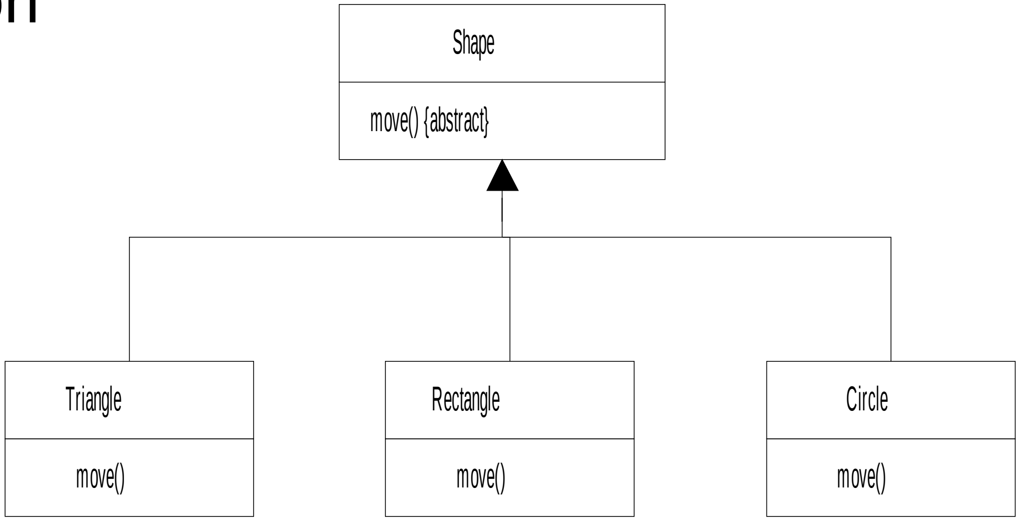
Interface Classes



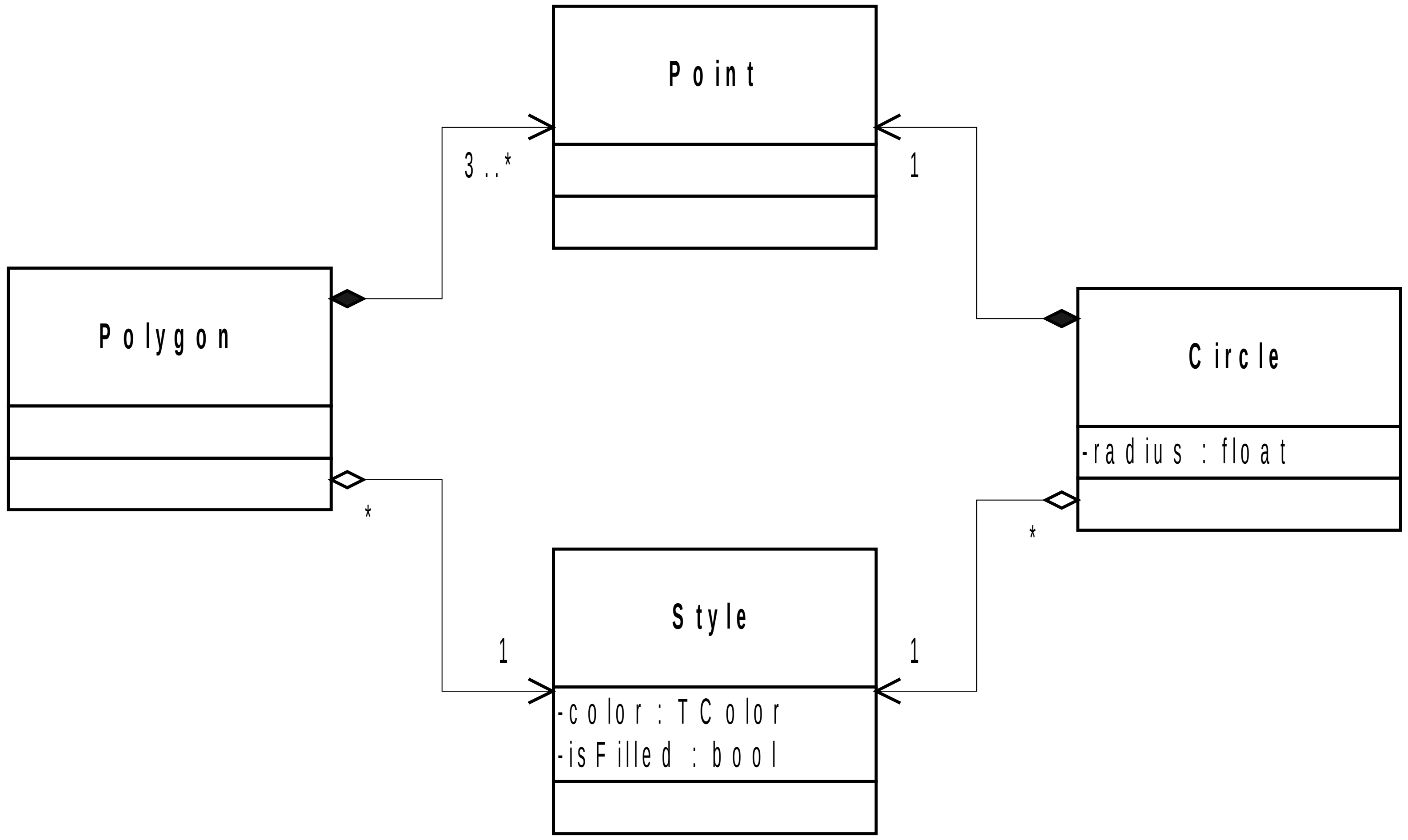
A class and an interface **differ**:
A class can have an actual instance of its type, whereas an interface must have at least one class to implement it. In UML 2, an interface is considered to be a specialization of a class modelling element. Therefore, an interface is drawn just like a class, but the top compartment of the rectangle also has the text "«interface»", as shown in Figure.

Abstract Classes

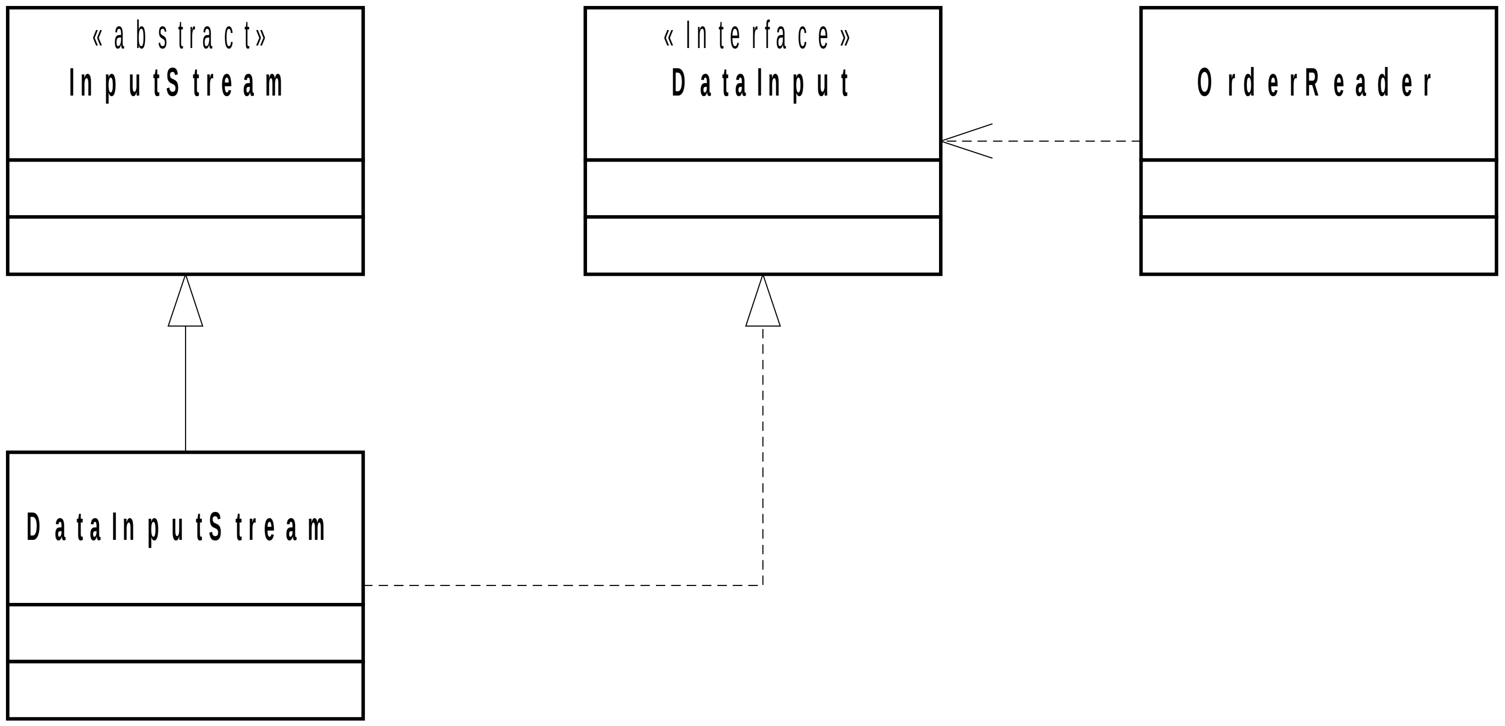
- A class that has no direct instances but whose descendants have direct instances
- The abstract class does not have a direct meaning
- The abstract class only has a meaning as an abstraction



Aggregation versus Composition



Interfaces and Abstract Classes



Fowler Chapters 4 and 11

Interaction Diagrams

Different Types of Interaction Diagrams

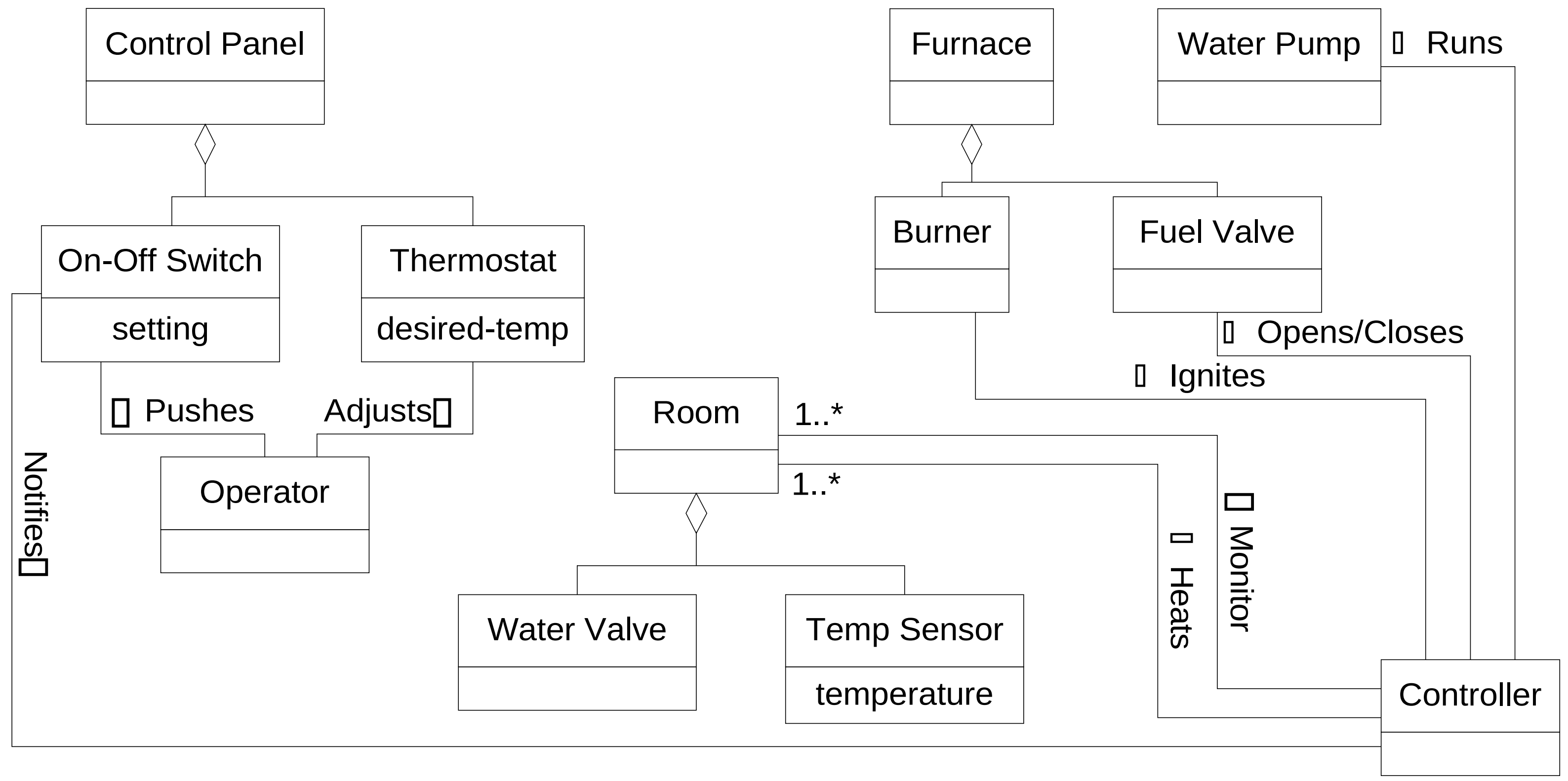
- An Interaction Diagram typically captures a use-case
  - A sequence of user interactions

- Sequence diagrams
  - Highlight the sequencing of the interactions between objects
- Collaboration diagrams
  - Highlight the structure of the components (objects) involved in the interaction

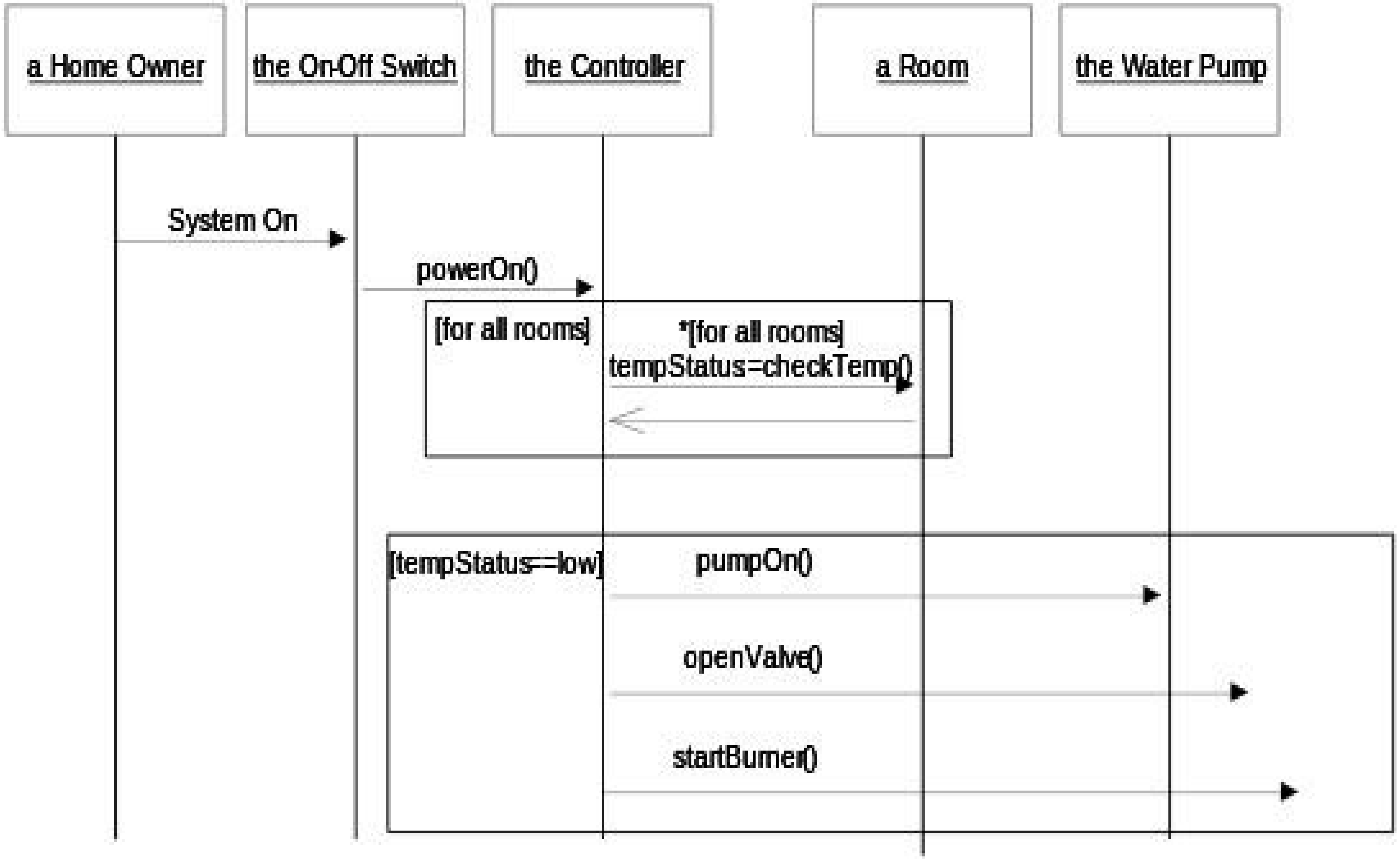
Home Heating Use-Case

Use case:	**Power Up**
Actors:	Home Owner (initiator)
Type:	Primary and essential
Description:	The Home Owner turns the power on. Each room is temperature checked. If a room is below the the desired temperature the valve for the room is opened, the water pump started, the fuel valve opened, and the burner ignited. If the temperature in all rooms is above the desired temperature, no actions are taken.
Cross Ref.:	Requirements XX, YY, and ZZ
Use-Cases:	None

Class Diagram—v1



Sequence Diagrams (for cd—v1)



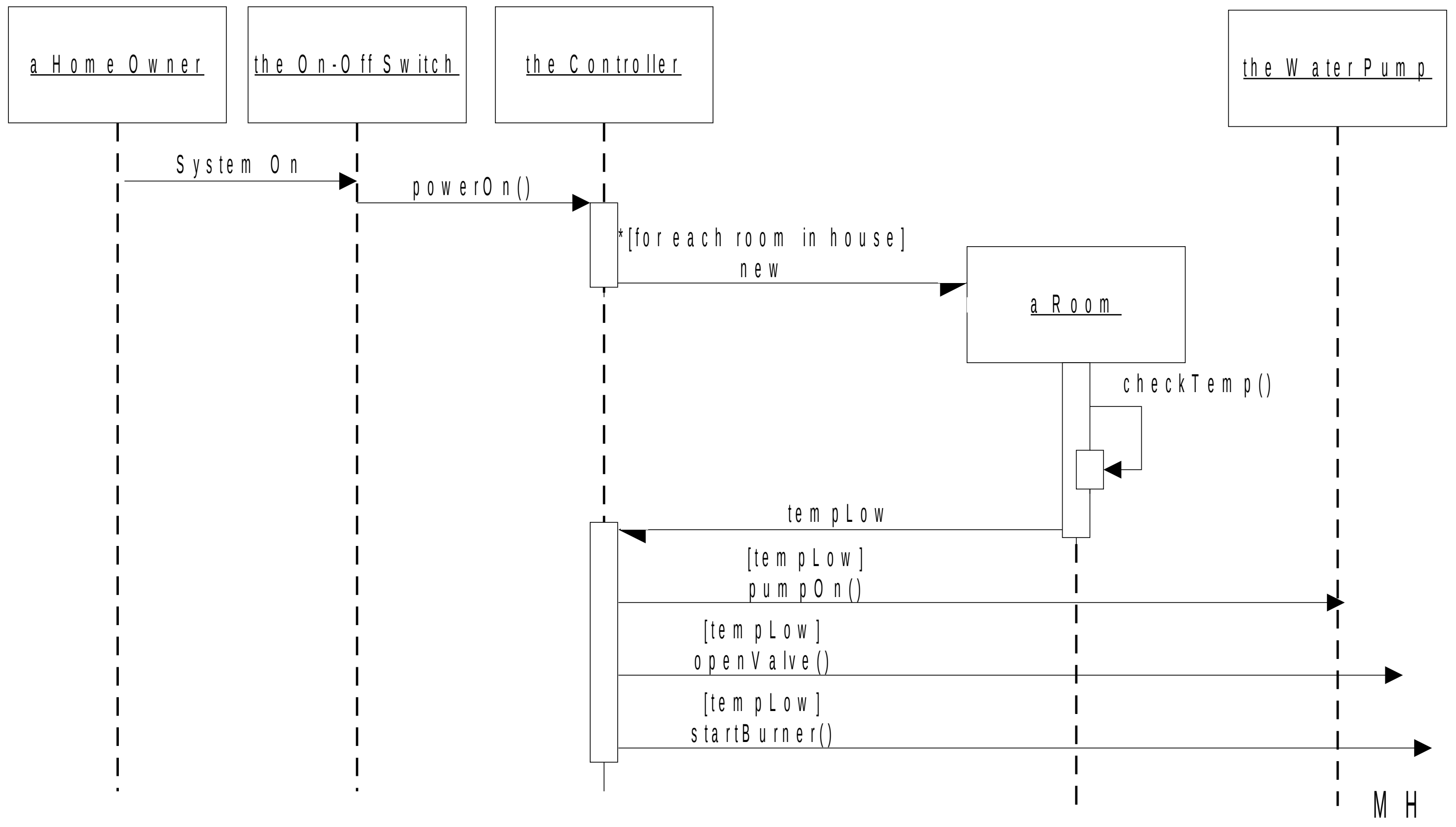
Comment the Diagram (for cd —v1)

When the owner turns the system on

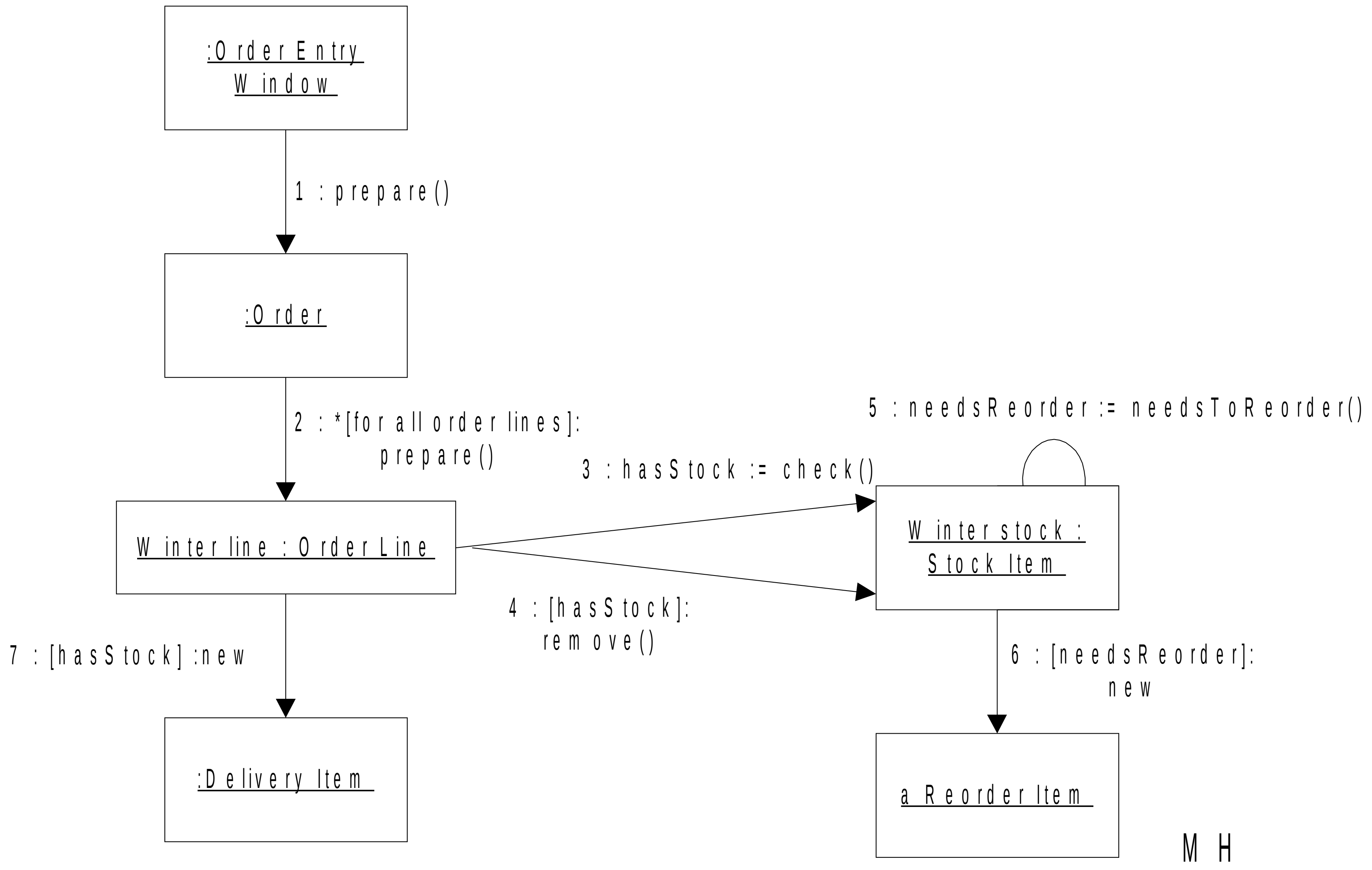
the on switch notifies the controller

The controller creates a room object for each room in the building

The rooms sample the temperature in the toom every 5 s. When a low temp is detected the room notifies the controller.



Collaboration Diagrams



Collaboration Diagrams Summary

- Highlights the structure of the components (objects) involved in the interaction
  - Better shows how the various objects are related to each other
  - Can help you identify which classes to put in a larger module
- Does the same thing as a sequence diagram, but with a different focus
- Again, clarity is the goal – use comments

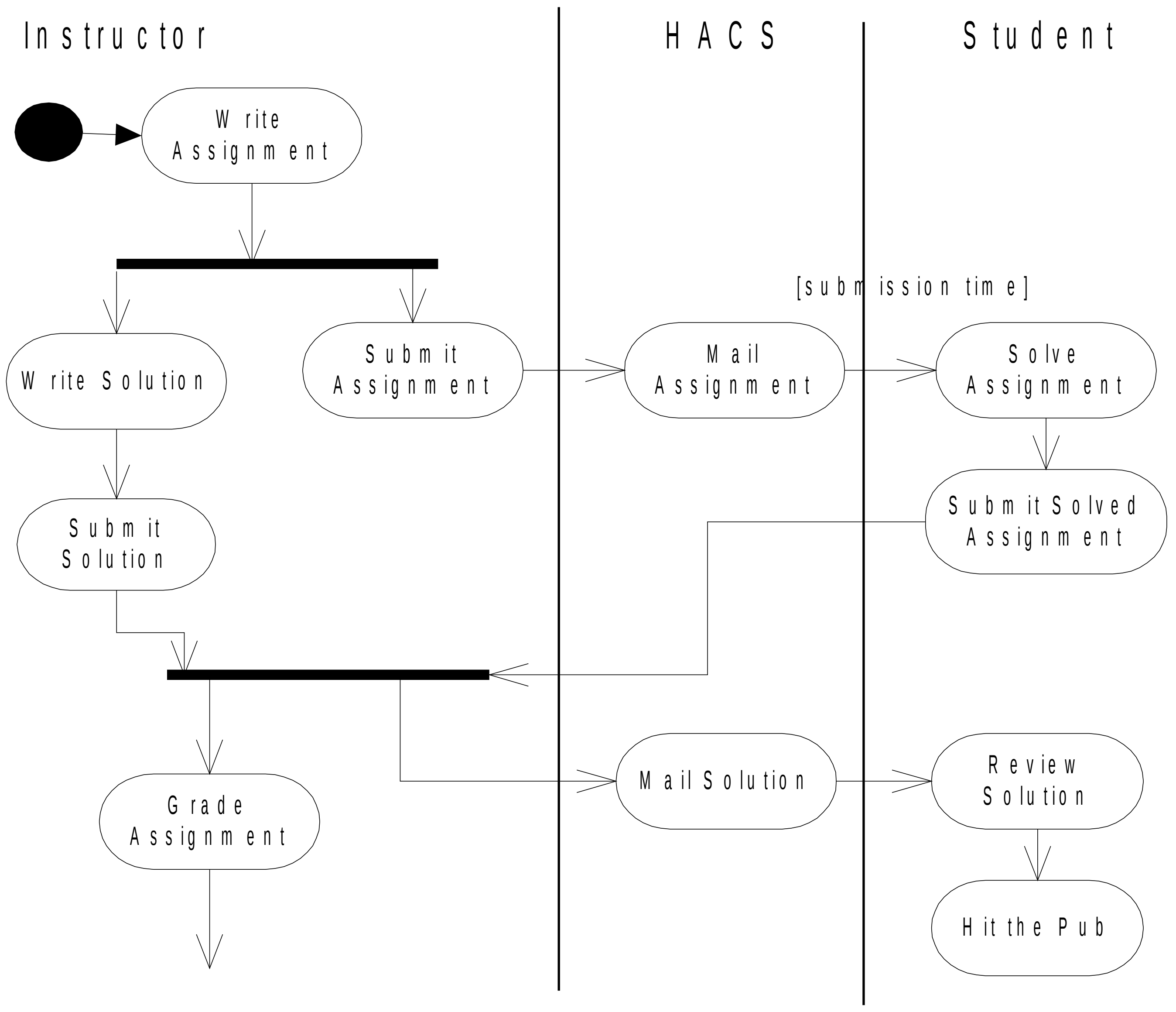
When to Use Interaction Diagrams

- When you want to clarify and explore single use-cases involving several objects
  - Quickly becomes unruly if you do not watch it
- If you are interested in one object over many use-cases — **state transition diagrams**
- If you are interested in many objects over many use cases — **activity diagrams**

Activity Diagrams

- Shows how activities are connected together
  - Shows the order of processing
  - Captures parallelism
- Mechanisms to express
  - Processing
  - Synchronization
  - Conditional selection of processing

Swimlanes (Who Does What?)



Problems with Activity Diagrams

- They are glorified flowcharts
  - Very easy to make a traditional data-flow oriented design
- Switching to the OO paradigm is hard enough as it is
  - Extensive use of activity charts can make this shift even harder
- However….
  - Very powerful when you know how to use them correctly

Fowler, Chapter 10

State Diagrams

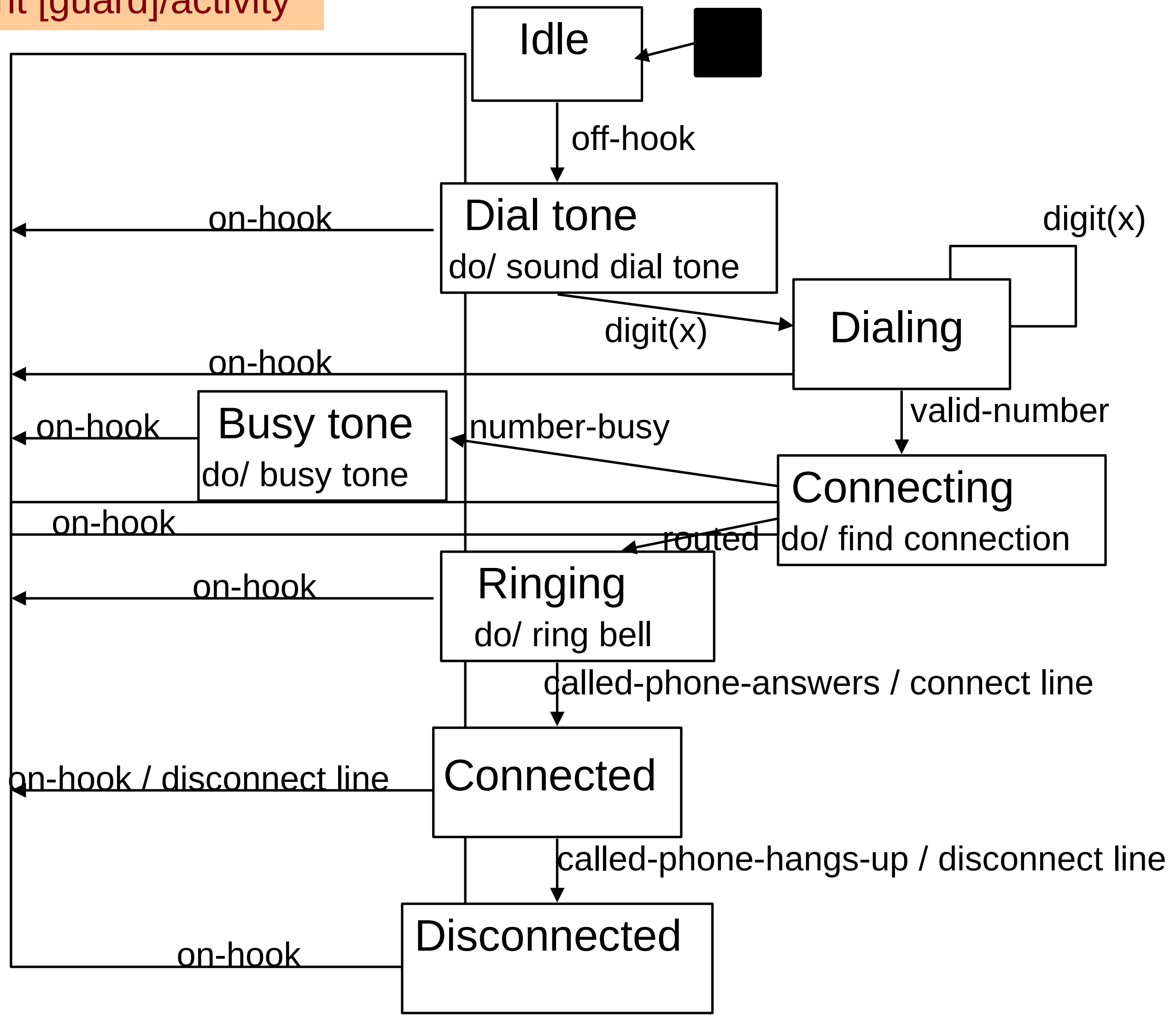
Events, Conditions, and States

- Event
  - Something that happens at a point in time
  - Operator presses self-test button
  - The alarm goes off
- Condition
  - Something that has a duration
  - The fuel level is high
  - The alarm is on
- State
  - An abstraction of the attributes and links of an object (or entire system)
  - The controller is in the state self-test after the self-test button has been pressed and the rest-button has not yet been pressed
  - The tank is in the state too-low when the fuel level has been below level-low for alarm-threshold seconds

Operations (AKA Actions)

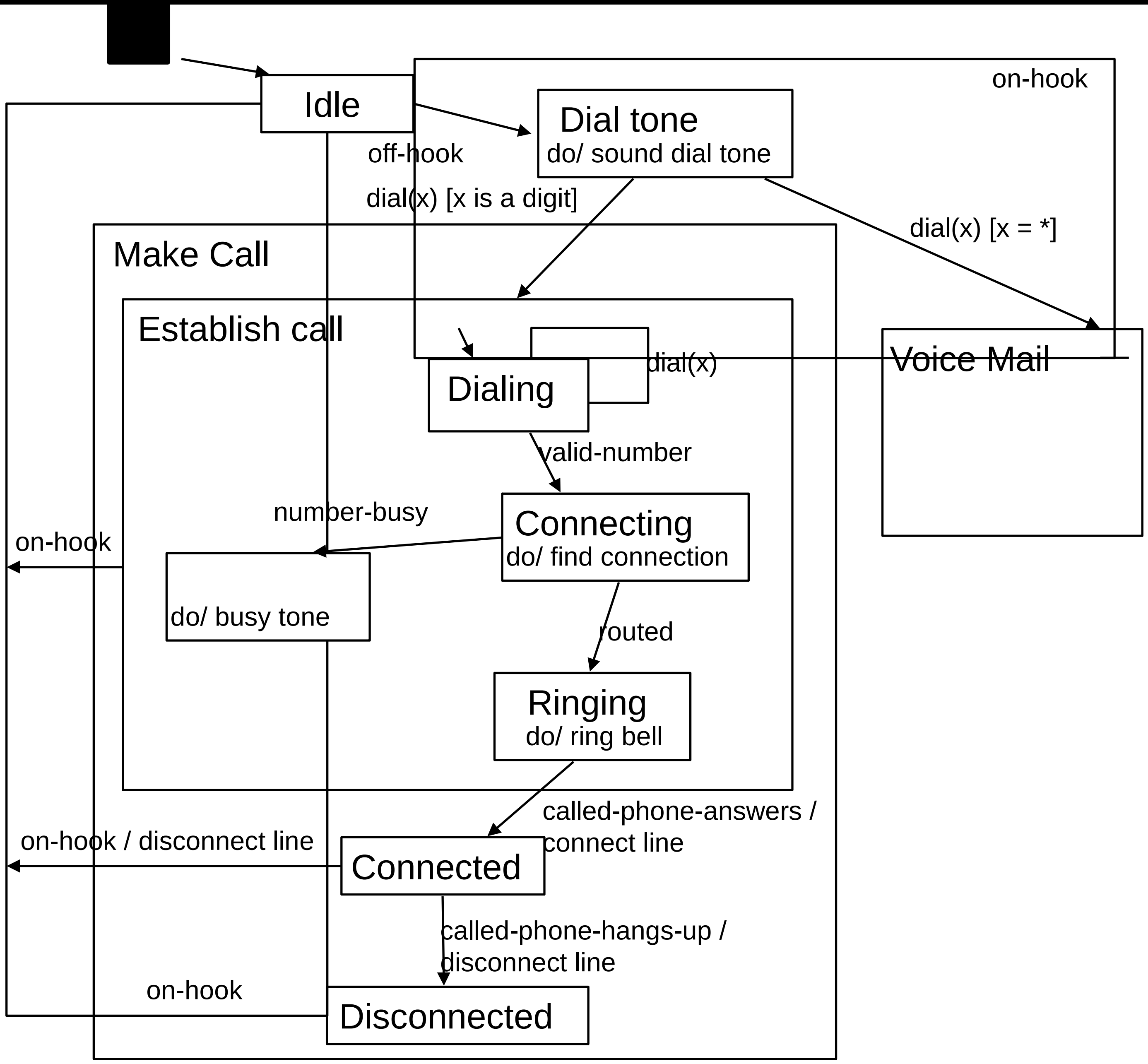
Transition label: trigger-event [guard]/activity

- Actions are performed when a transition is taken or performed while in a state
- Actions are terminated when leaving the state



Hierarchical State Machines

- Group states with similar characteristics
- Enables information hiding
- Simplifies the diagrams



Design Patterns

Slides courtesy Gregory Gay

Guidelines, not solutions

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

- Christopher Alexander

Categories of design patterns

1. Creational

   Decouple a client from objects it instantiates.

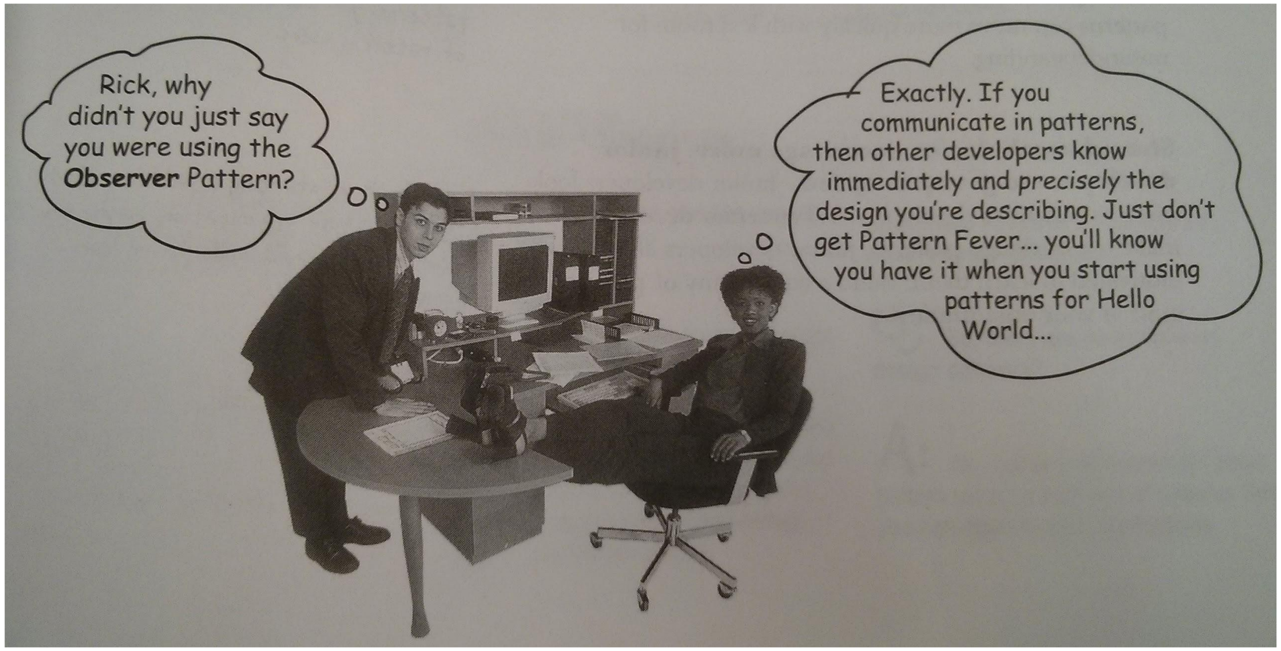
2. Structural

   Clean organization into subsystems.

3. Behavioral

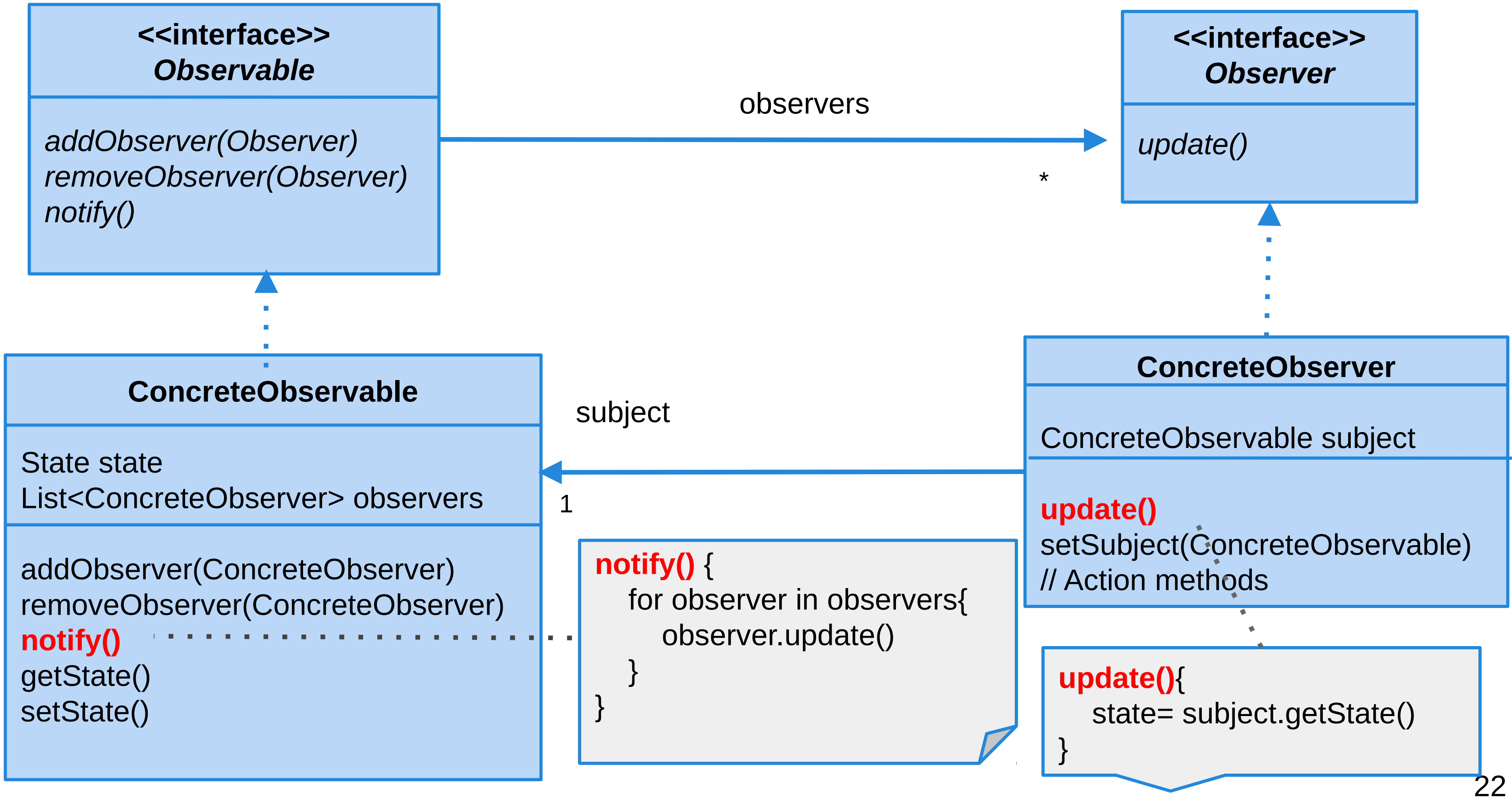
   Describe how objects interact.

Why use design patterns?


Rick, why didn't you just say you were using the **Observer** Pattern?
Exactly. If you communicate in patterns, then other developers know immediately and *precisely* the design you're describing. Just don't get Pattern Fever... you'll know you have it when you start using patterns for Hello World...

1. Good examples of OO principles.
2. Faster design phase.
3. Evidence that system will support change.
4. Offers shared vocabulary between designers.

Observer Pattern - In Practice



```
notify() {
    for observer in observers{
        observer.update()
    }
}
```

```
update(){
    state= subject.getState()
}
```

Why not use a design pattern?

What are the drawbacks to using patterns?

- Potentially over-engineered solution.
- Increased system complexity.
- Design inefficiency.

How can we avoid these pitfalls?

Sommerville Chapter 6

The High-Level Structure of a Software Intensive System

Architectural Design

What is Architecture informally?

- Software architecture is primarily concerned with **partitioning** large systems into smaller ones that can be created separately, that individually have business value, and that can be straightforwardly integrated with one another and with existing systems.

Mike Whalen

Architectural Design Process

- System structuring
  - The system is decomposed into several principal sub-systems and communications between these sub-systems are identified
- Control modeling
  - A model of the control relationships between the different parts of the system is established
- Modular decomposition
  - The identified sub-systems are decomposed into modules

Architectural Qualities

Performance

Ease of
Maintenance

Testability

Security

Usability

Key Points

- The software architect is responsible for deriving a structural system model, a control model and a sub-system decomposition model
- Large systems rarely conform to a single architectural model
- **System decomposition models** include repository models, client-server models and abstract machine models
- **Control models** include centralized control and event-driven models

Key Points

- Modular decomposition models include data-flow and object models
- Domain specific architectural models are abstractions over an application domain
  - They may be constructed by abstracting from existing systems or may be idealized reference models

Sommerville Chapter 8

Software Testing: Definitions and Fundamentals

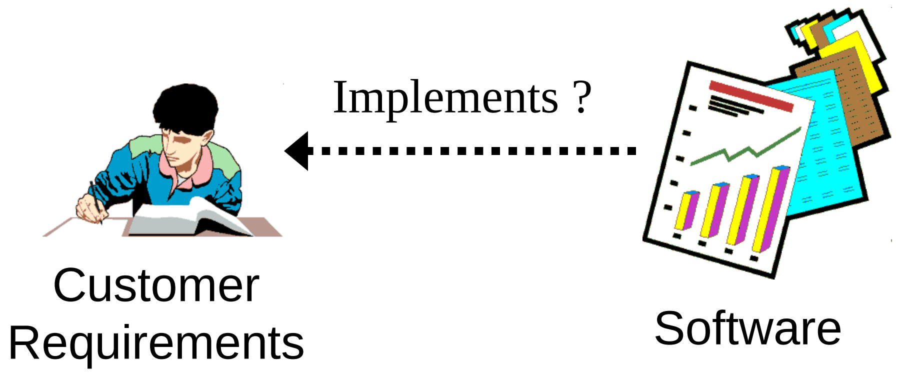
Slides from Prof. Mats Heimdahl

Verification and Validation: IEEE

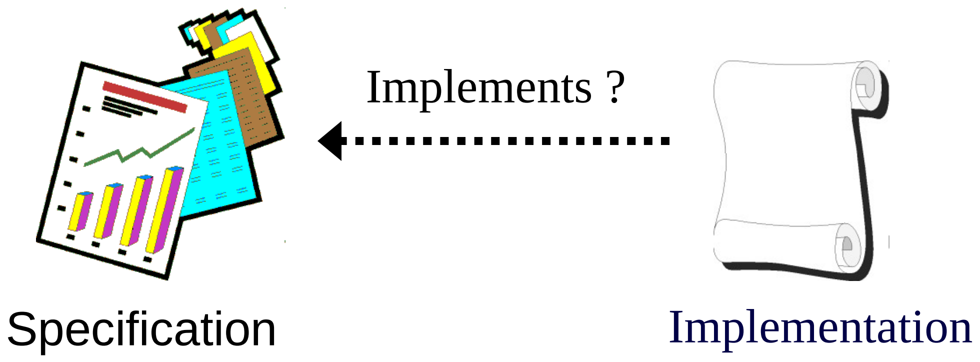
- Verification
  - The process of evaluating a system or component to determine whether the products…satisfy the conditions imposed…
- Validation
  - The process of evaluating a system or component…to determine whether it satisfies specified requirements.

Validation and Verification

Validation: Are we building the right product?



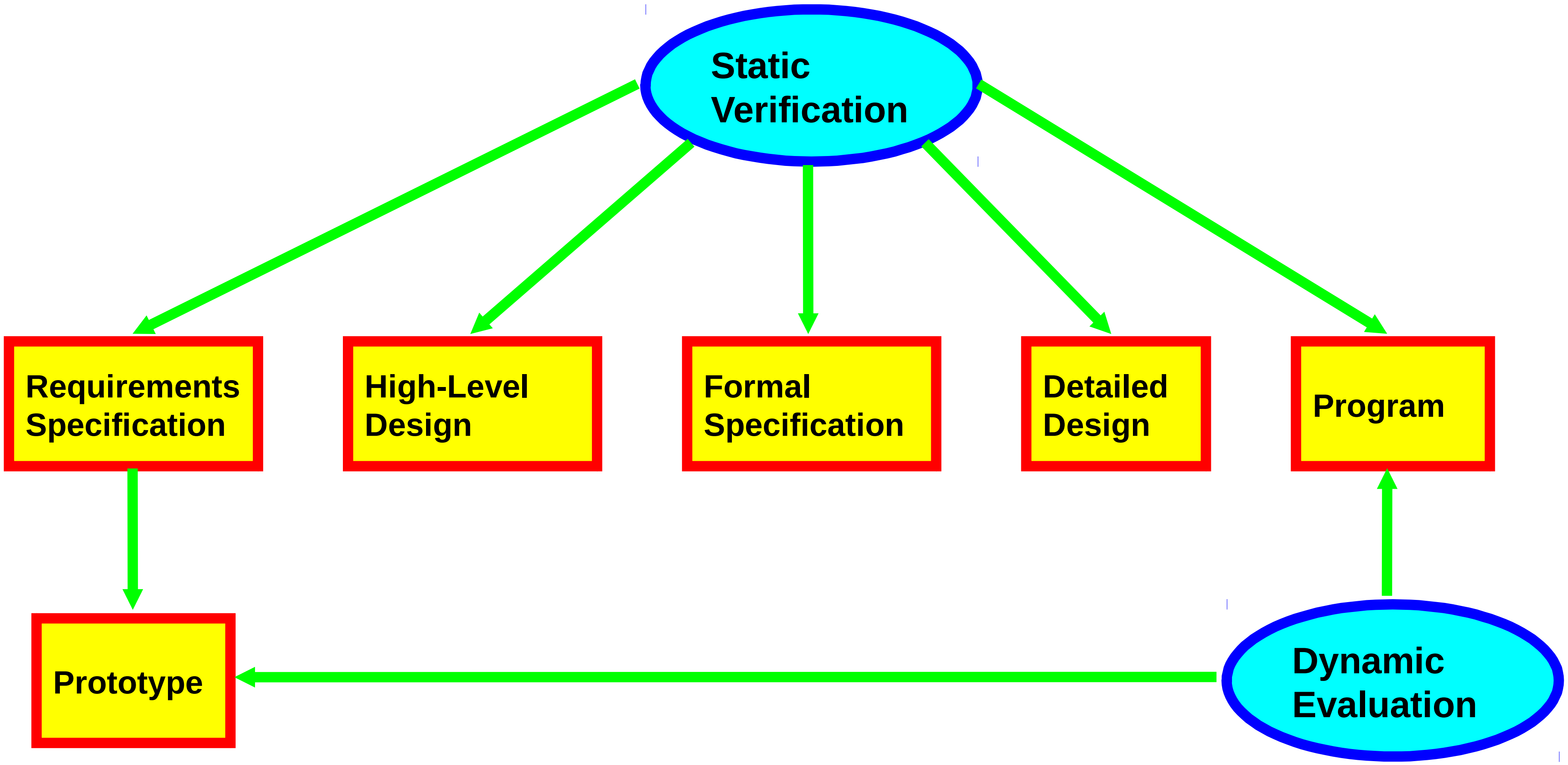
Verification: Are we building the product right?



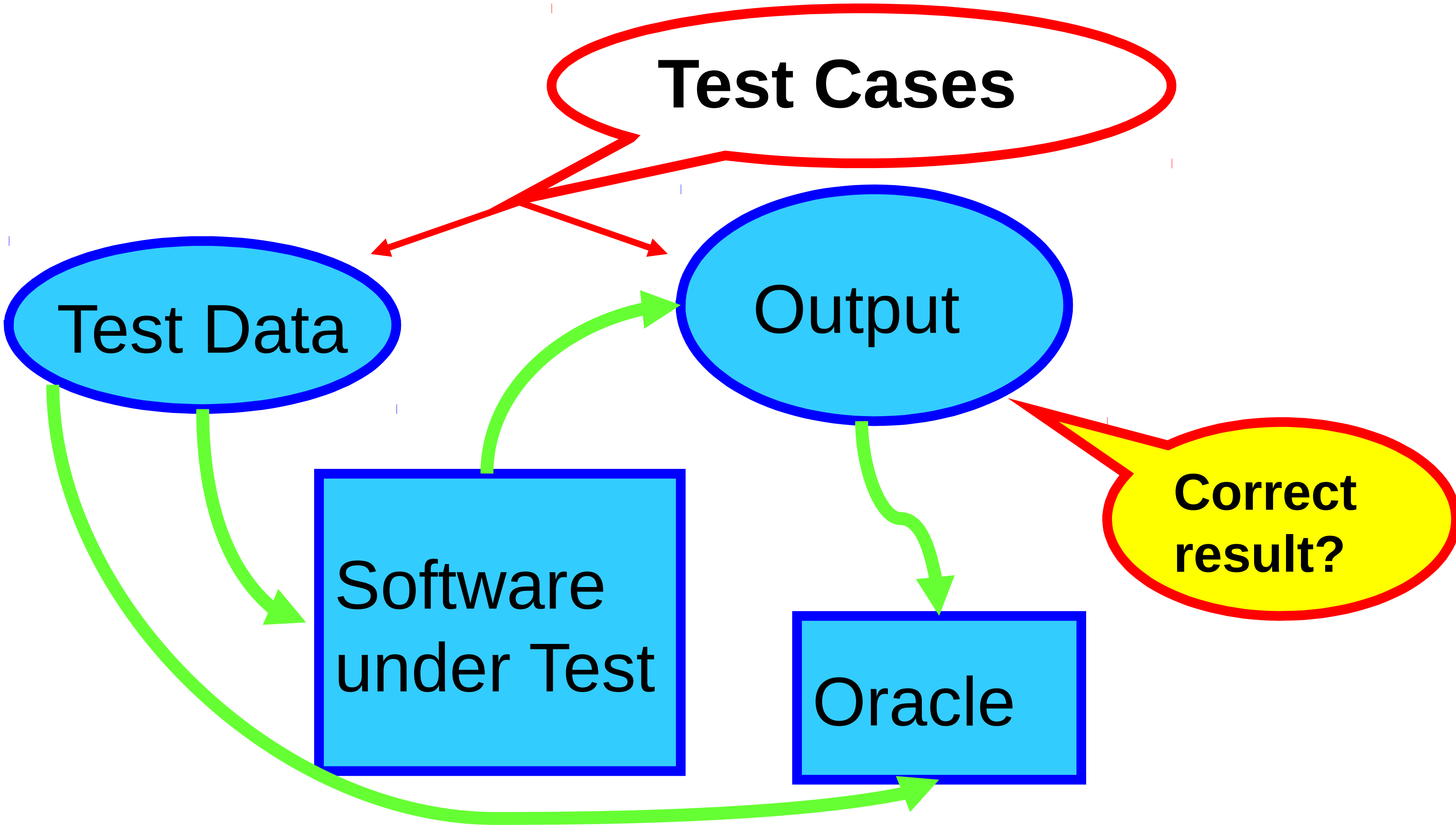
Dynamic and Static Verification

- Dynamic V & V
  - Concerned with exercising and observing product behavior
  - Testing
- Static V & V
  - Concerned with analysis of the static system representation to discover problems
  - Proofs
  - Inspections

Static and Dynamic V&V



What is a Test?



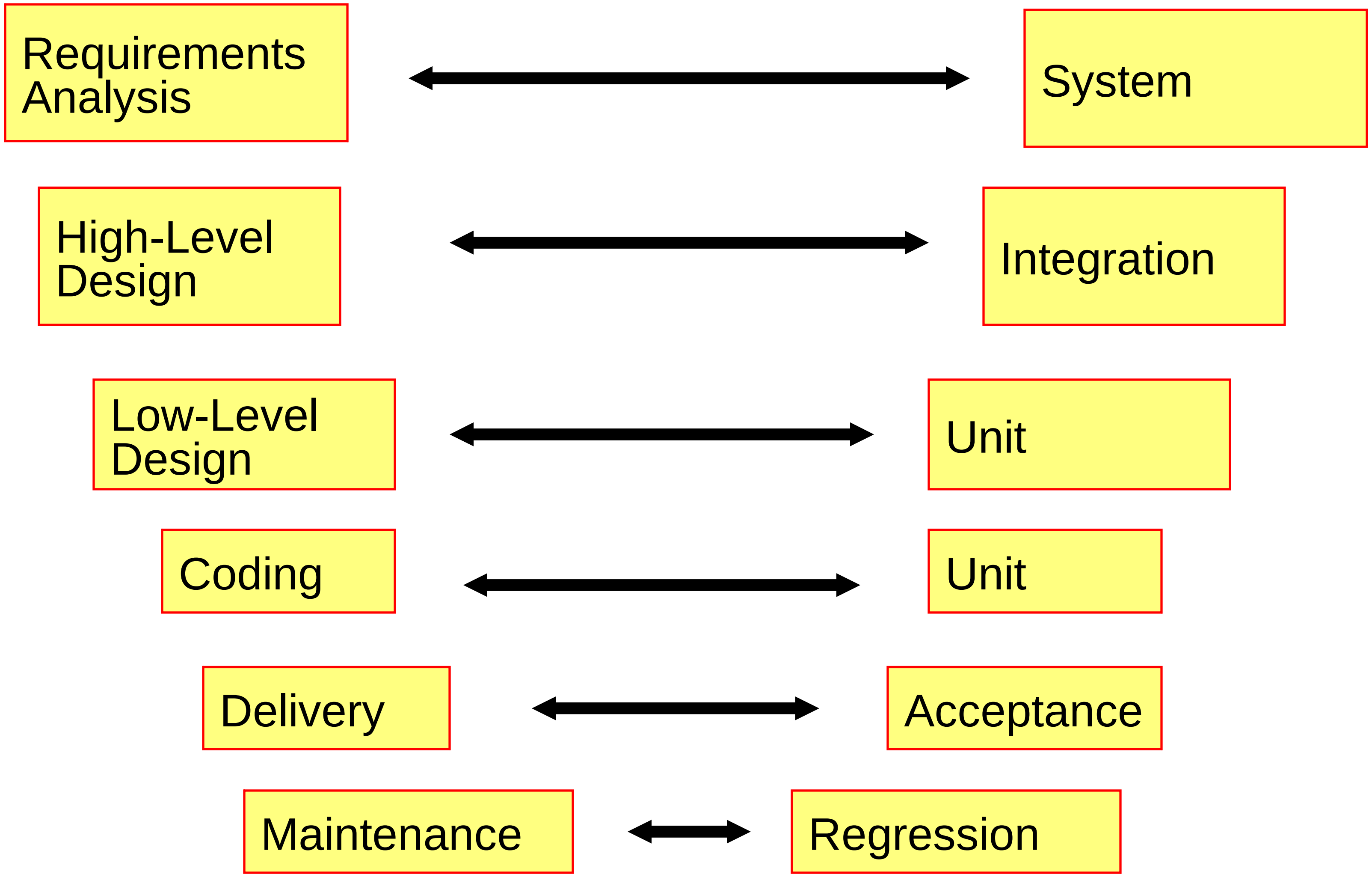
Bugs? What is That?

- Failure
  - An execution that yields an erroneous result
- Fault
  - The source of the failure
- Error
  - The mistake that led to the fault being introduced in the code

Testing Stages

- Unit testing
  - Testing of individual components
- Module testing
  - Testing of collections of dependent components
- Sub-system testing
  - Testing collections of modules integrated into sub-systems
- System testing
  - Testing the complete system prior to delivery
- Acceptance testing
  - Testing by users to check that the system satisfies requirements
  - Sometimes called alpha and beta testing

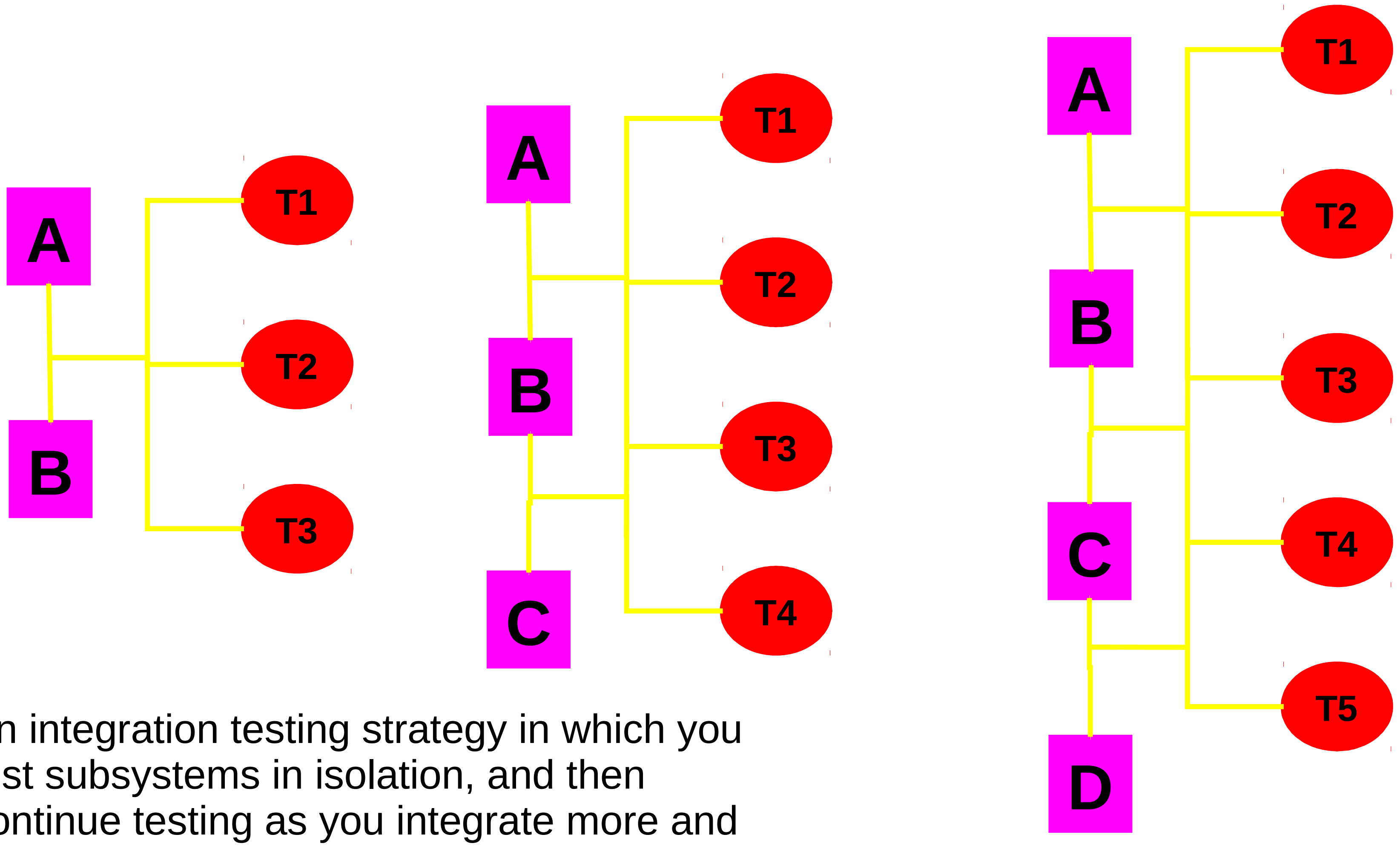
V Graph


Requirements Analysis
System
High-Level Design
Integration
Low-Level Design
Unit
Coding
Unit
Delivery
Acceptance
Maintenance
Regression

Testing Strategies

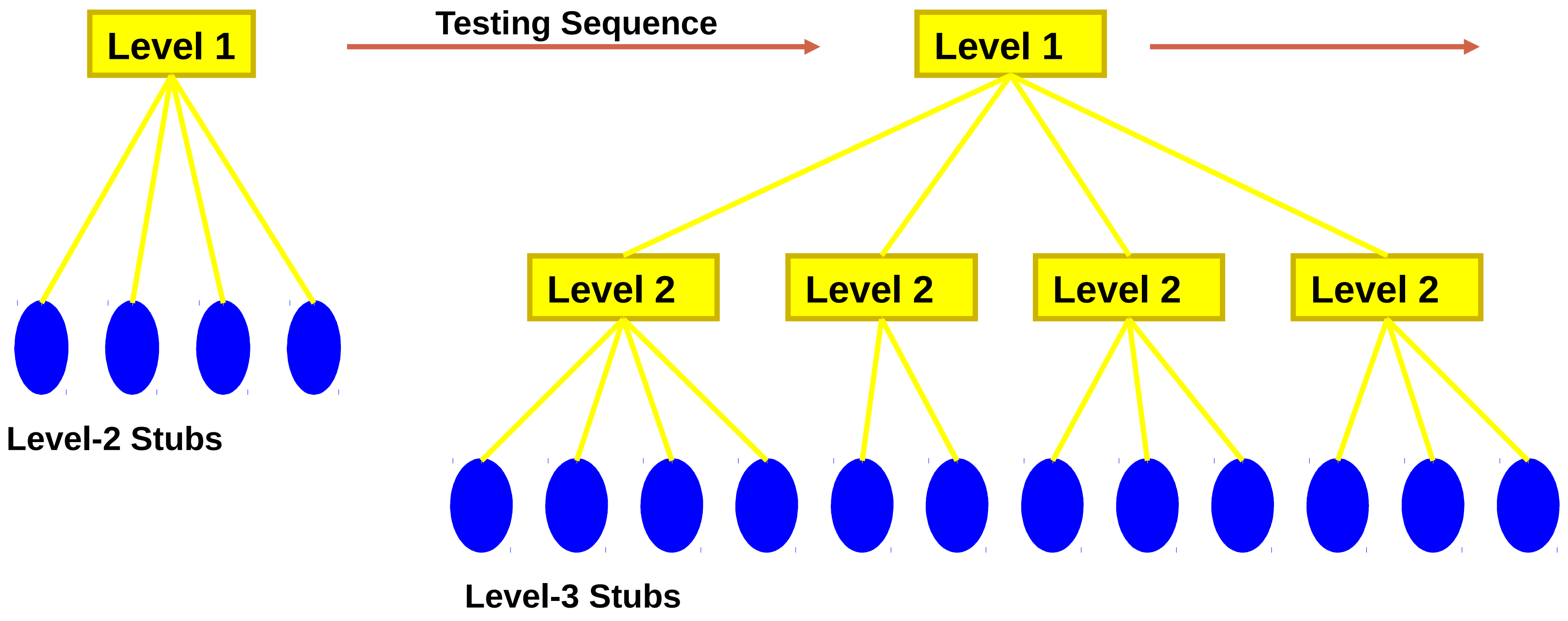
- Testing strategies are ways of approaching the testing process
- Different strategies may be applied at different stages of the testing process
- Strategies covered
  - Top-down testing
  - Bottom-up testing
  - Back-to-back testing

Incremental Testing

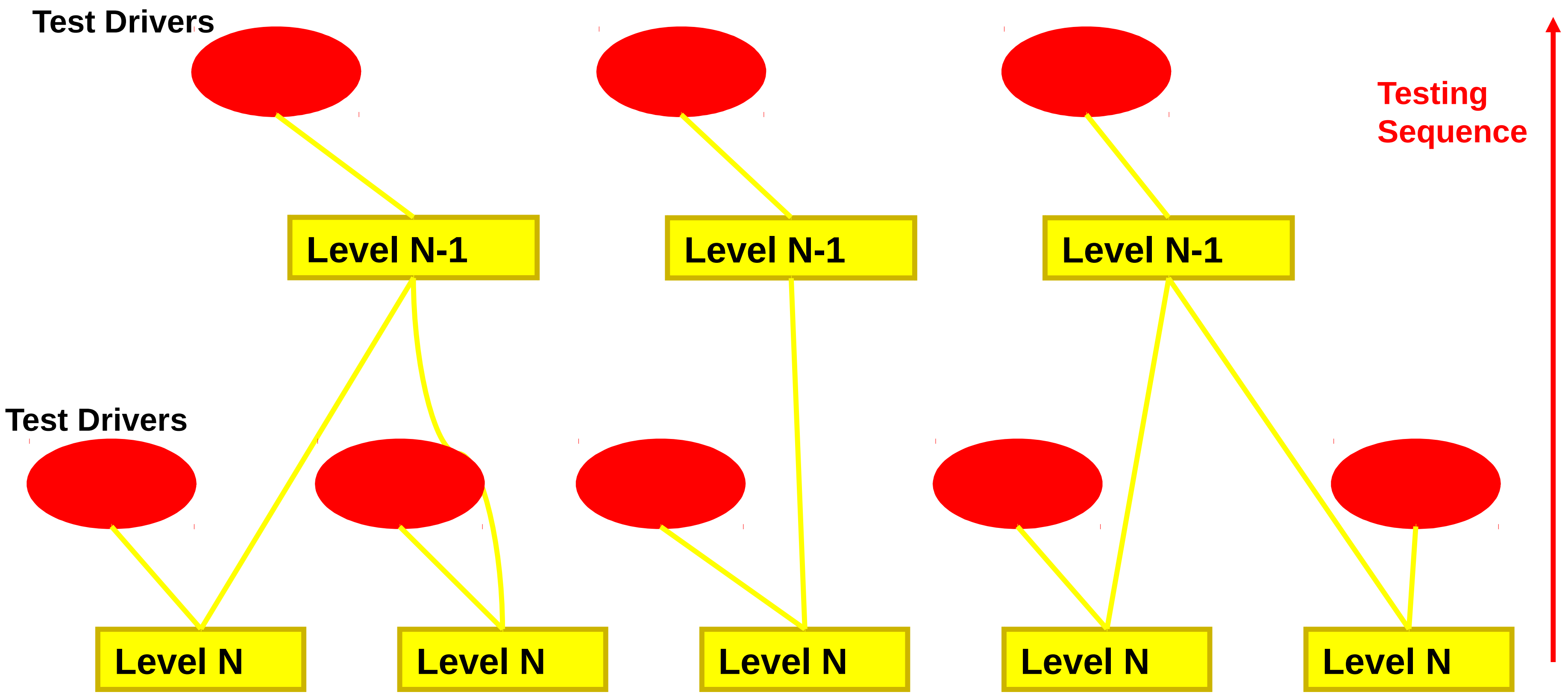


An integration testing strategy in which you test subsystems in isolation, and then continue testing as you integrate more and more subsystems

Top-down testing



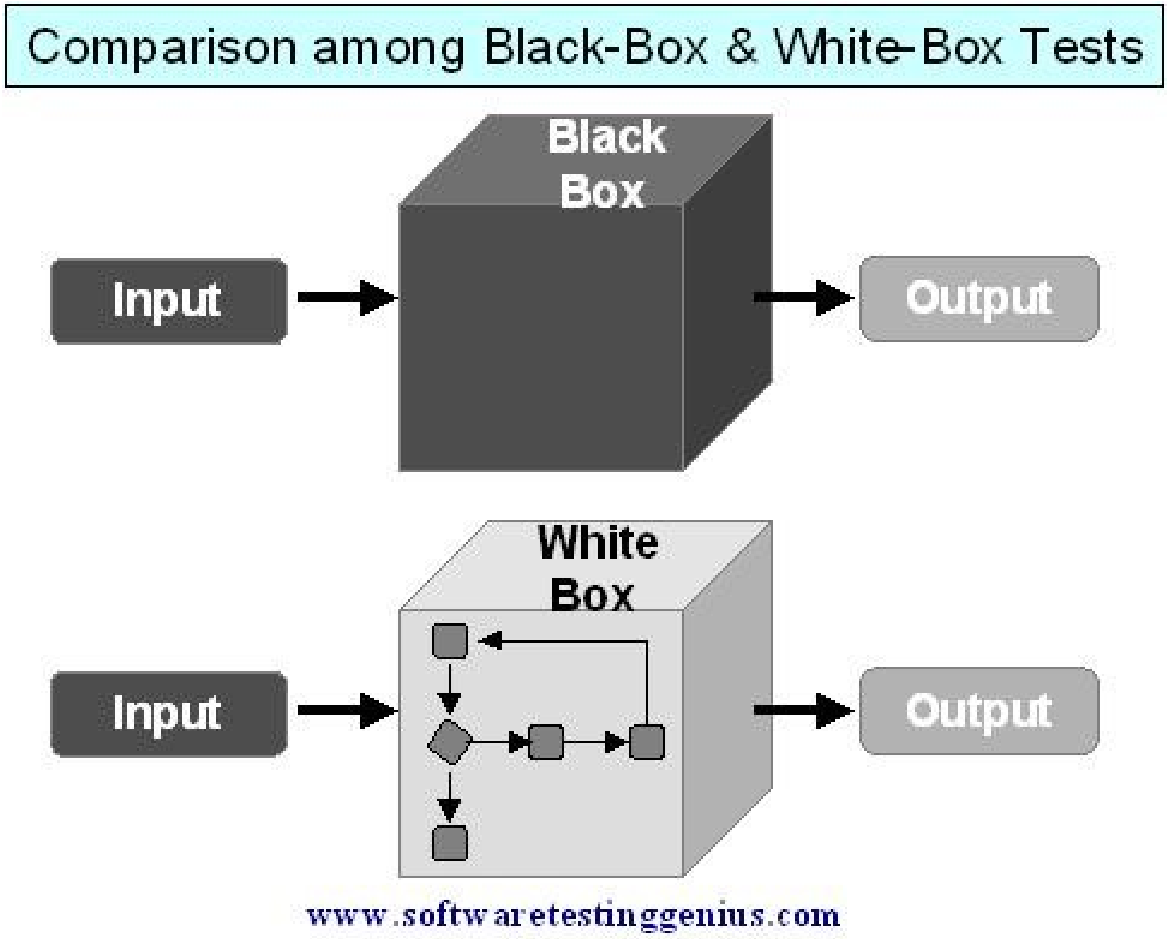
Bottom-Up Testing



Sommerville Chapter 8
(we will come back here later)

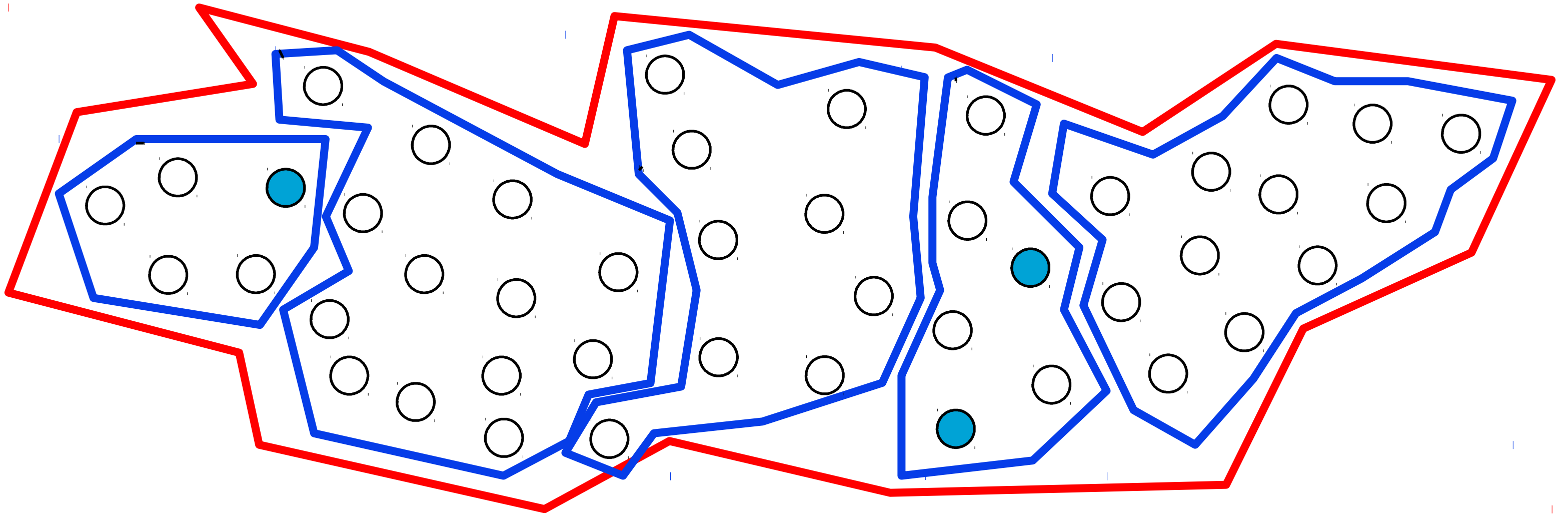
Software Testing: Requirements Based (Black box)

Black and White Box


Comparison among Black-Box & White-Box Tests
Black Box
Input
Output
White Box
Input
Output
www.softwaretestinggenius.com

Partition Testing

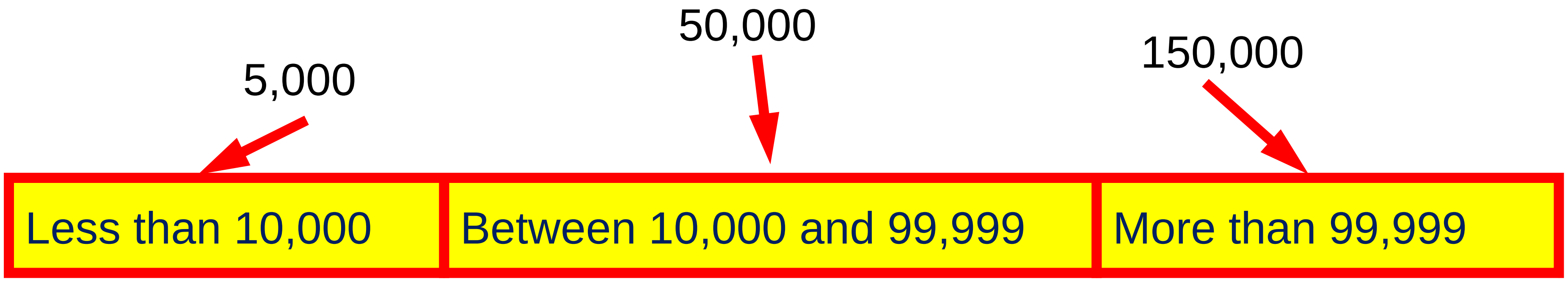
- Basic idea: Divide program input space into (quasi-) equivalence classes
  - Underlying idea of specification-based, structural, and fault-based testing



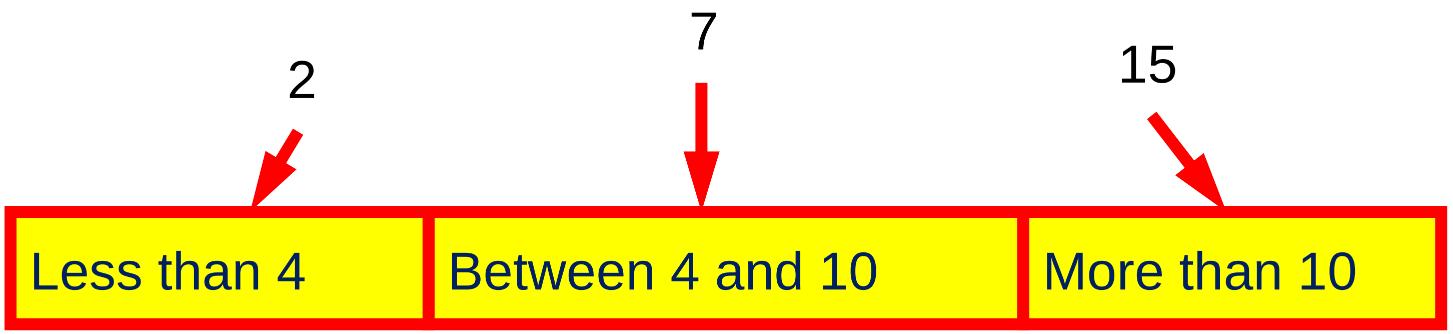
Equivalence Class?

- A group of tests form an equivalence class if
  - They all test the same thing
  - If one test reveals a fault, the other ones (probably) will too
  - If a test does not reveal a fault, the other ones (probably) will not either

Equivalence Partitions

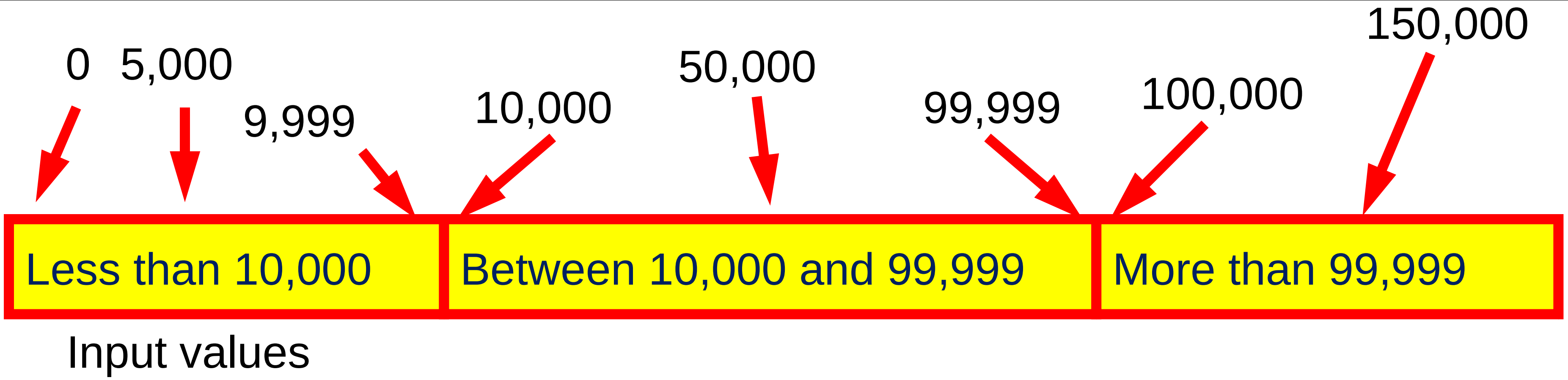


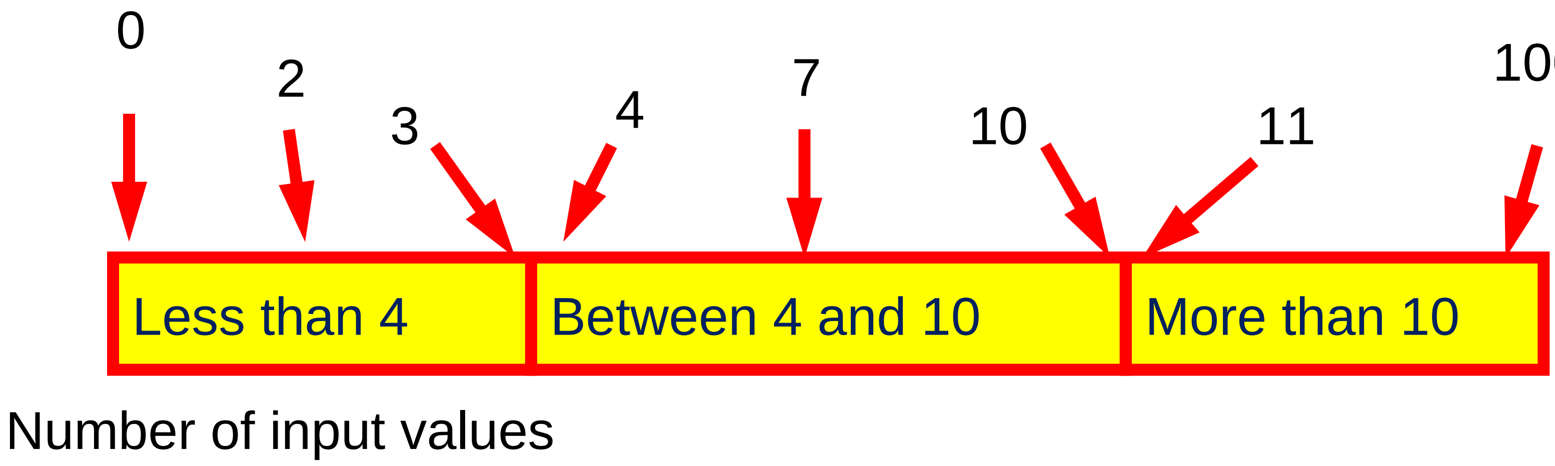
Input values



Number of input values

Equivalence Partitions Revisited





Do Not Forget Invalid Inputs!

- Most likely to cause problems
  - Exception handling is a well know problem area
  - People tend to think about what the program shall do, not what it shall protect itself against

- Take this into account with all selection criteria we have discussed this far

Using the code to measure test adequacy
(and derive test cases)

Structural Testing

Structural Testing

- Sometime called white-box testing
- Derivation of test cases according to program structure
  - Knowledge of the program is used to identify test cases
- Objective is to exercise a certain percentage of statements, branches, or condition (not all path combinations)
  - Why??

Program Flow Graphs

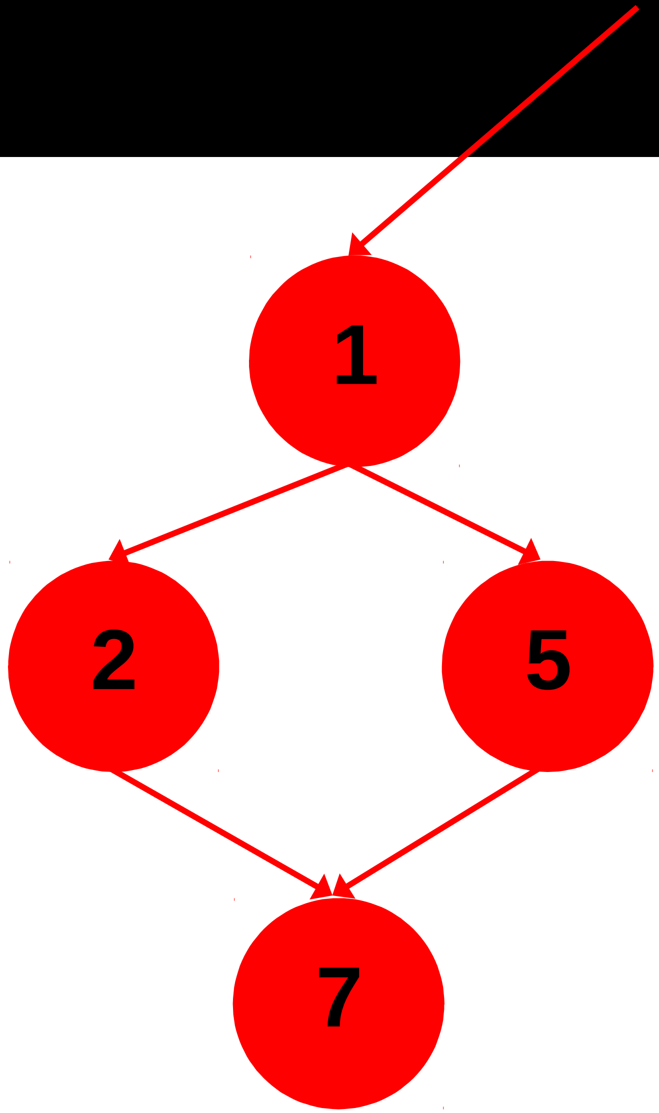
- Describes the program control flow
- Used as a basis for test data selection
- Used as a basis for computing the cyclomatic complexity
- Complexity = Number of edges - Number of nodes +1
  - Number of decision points + 1
  - N way branch counts as N-1 decision points

Cyclomatic Complexity

- CC = E – N + 1 when every exit point is connected back to the entry point in the control flow graph.
- CC = E – N + 2 when exit point is **not** connected back to entry point

If-then-else

```
1 if (1==x) {
2 y=45;
3 }
4  else {
5 y=23456;
6 }
7 /* continue */
```



Structural Coverage Testing

- (In)adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) coverage
  - Condition coverage
  - Path coverage
  - Data flow (syntactic dependency) coverage
- Attempted compromise between the impossible and the inadequate

A Program with a Bug

- This following program inputs an integer *x*
  - if *x* < 0, transforms it into a positive value before invoking foo-1 to compute the output *z*
  - if *x*≥0, compute z using foo-2

```
begin
  int x, z;
  input (x);
  if(x<0)
   x =-x;
  z=foo-1(x);
  output(z);
end
```

Where is the bug?

There should have been an else clause for *x*≥0 before this statement.

Is Statement Coverage Sufficient?

- Consider a test set $T=\{t_1:<x=-5>\}$.
- It is adequate with respect to *statement* coverage criterion, but does not reveal the bug.

```
begin
  int x, z;
  input (x);
  if(x<0)
   x =-x;
  z=foo-1(x);
  output(z);
end
```

There should have been an *else* clause for $x \geq 0$ before this statement.

Is Decision Coverage Sufficient?

- Consider another test set $T'=\{t_1{:}{<}x = -5{>}\ \ t_2{:}{<}x = 3{>}\}$
- T' is decision adequate, but not T.
- Also, T' reveals the bug, but not T.

```
begin
  int x, z;
  input (x);
  if(x<0)
    x =-x;
  z=foo-1(x);
  output(z);
end
```

There should have been an *else* clause for $x \geq 0$ before this statement.

- This example illustrates *how and why decision coverage might help in revealing a bug that is not revealed* by a test set adequate with respect to *statement coverage*.

We Have Learned

- Test Coverage Measures
  - Statement, branch, and path coverage
  - Condition coverage (basic, compound)
  - Data flow coverage
- Test coverage measures ensure that statements have been executed to some level
  - However, it is not possible to exercise all combinations

When have we tested enough?

When to Stop Testing

Today’s Topics

- How do we know when we are done?
- Stopping Criteria
  - Coverage
  - Budget
  - Plan
  - Reliability
  - Mutation analysis

Categorizing and specifying the reliability of software systems

Software Reliability

Courtesy Prof. Mats Heimdahl

Software Reliability

- Cannot be defined objectively
  - Reliability measurements which are quoted out of context are not meaningful
- Requires operational profile for its definition
  - The operational profile defines the expected pattern of software usage
- Must consider fault consequences
  - Not all faults are equally serious
  - System is perceived as more unreliable if there are more serious faults

Reliability Metrics

- Probability of failure on demand
  - This is a measure of the likelihood that the system will fail when a service request is made
  - POFOD = 0.001 means 1 out of 1000 service requests result in failure
- Rate of fault occurrence (ROCOF)
  - Frequency of occurrence of unexpected behavior
  - ROCOF of 0.02 means 2 failures are likely in each 100 operational time units

Reliability Metrics

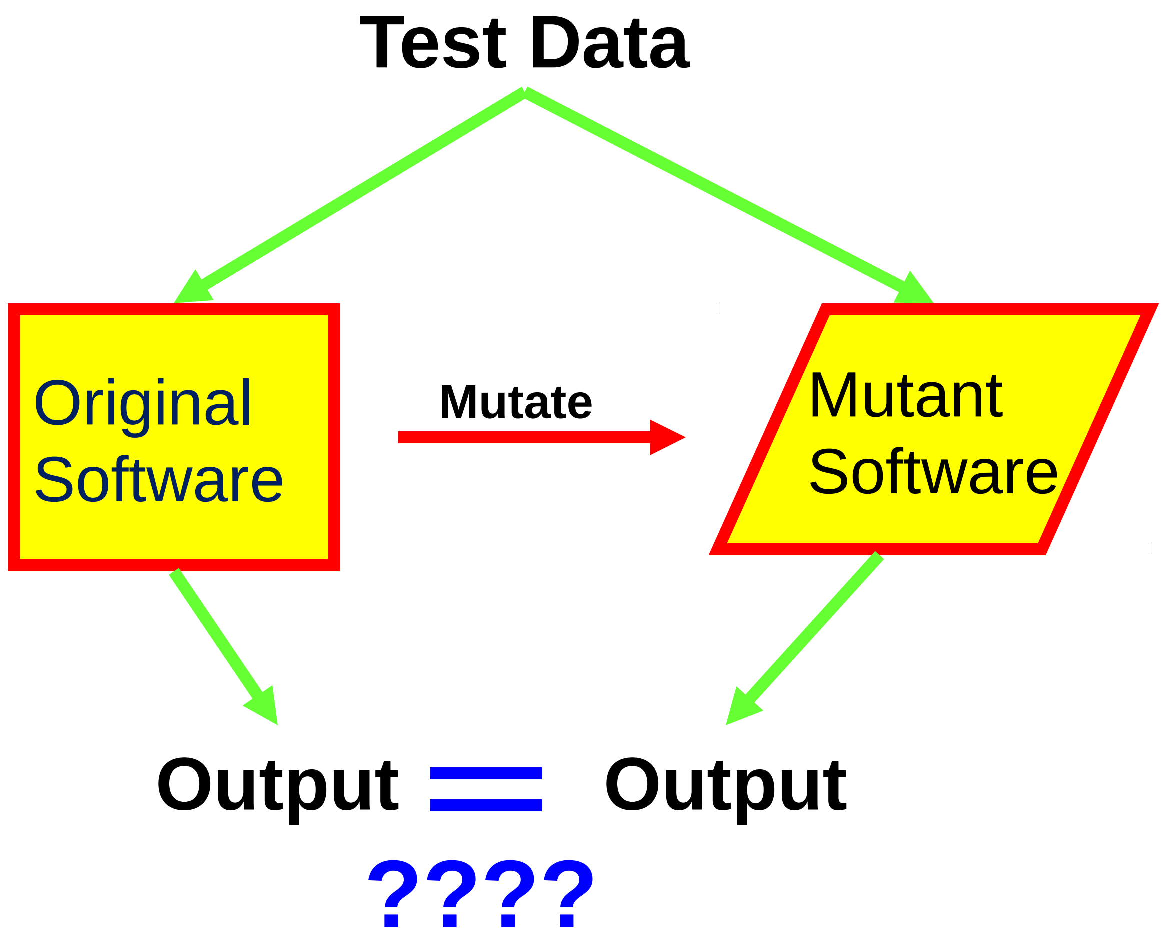
- Mean time to failure
  - Measure of the time between observed failures
  - MTTF of 500 means that the time between failures is 500 time units
- Availability
  - Measure of how likely the system is available for use. Takes repair/restart time into account
  - Availability of 0.998 means software is available for 998 out of 1000 time units

Mutation Testing

- An approach to investigating the quality of your test data

- Create a second version of your software with some minor change
  - Introduce a “mutation”
- Run the test cases and see if they reveal the mutation (an artificial fault)
  - If yes – Good test data
  - If no – Bad test data

General Idea


Test Data
Original Software
Mutate
Mutant Software
Output
Output
????