Software Requirements to Design

Announcements

- HW1 – Any questions?
- Difference between Requirements and Use Cases -
  - Requirements: Functional requirements capture the intended behavior of the system. This behavior may be expressed as services, tasks or functions the system is required to perform.
  - Use Cases: A use case defines a goal-oriented set of interactions between external actors and the system under consideration.
- HW1 due Thursday at Noon
- HW2 handed out on Thursday

Mainly “Will It Work?”

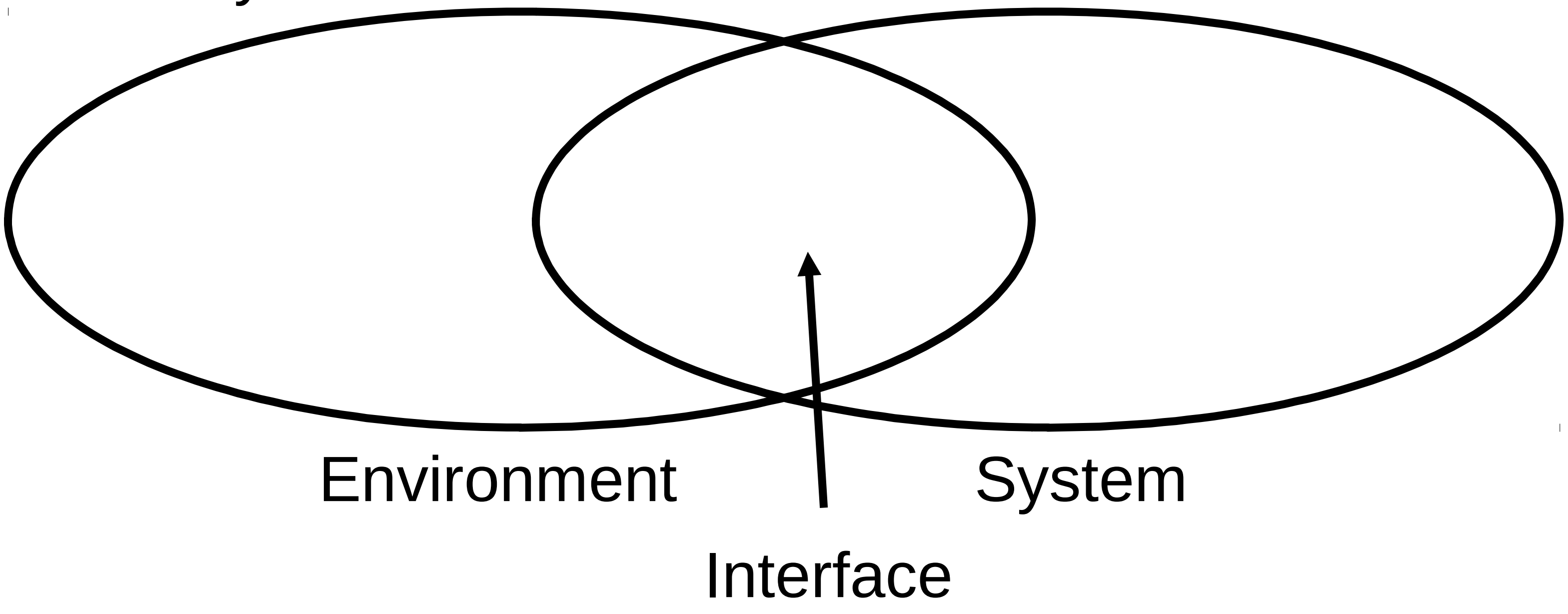
The World Machine Model

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

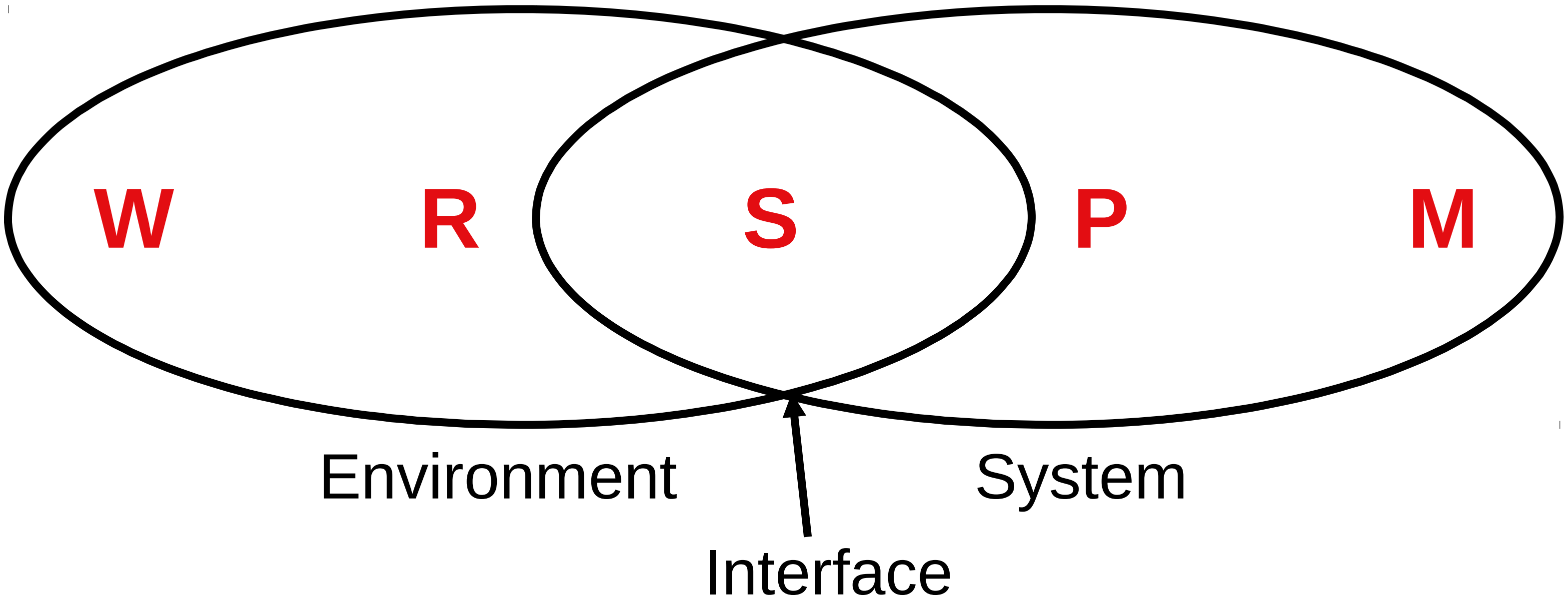
The WRSPM Model

- We want to make a change in the environment
- We will build some system to do it
- This system must interact with the environment

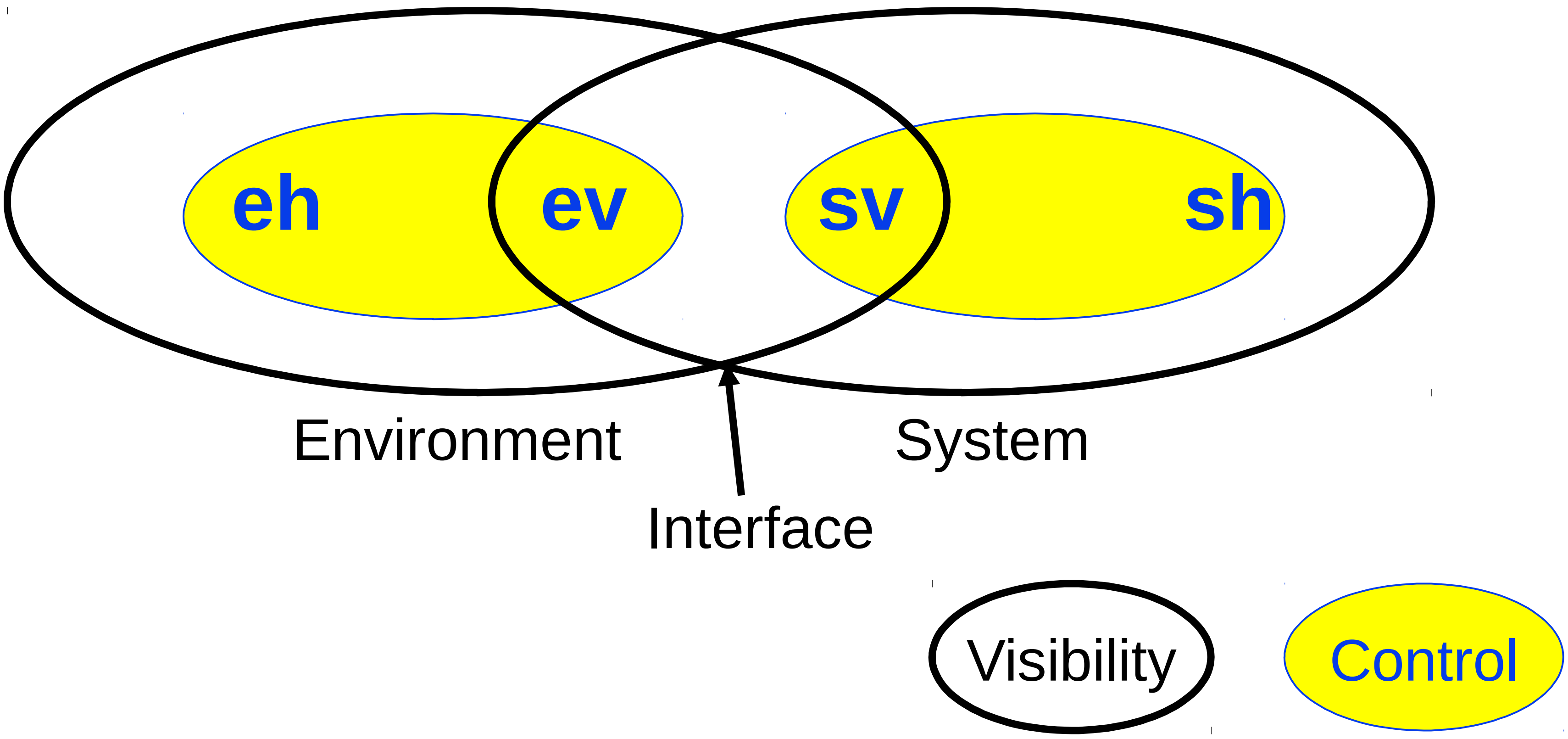


The WRSPM Model

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

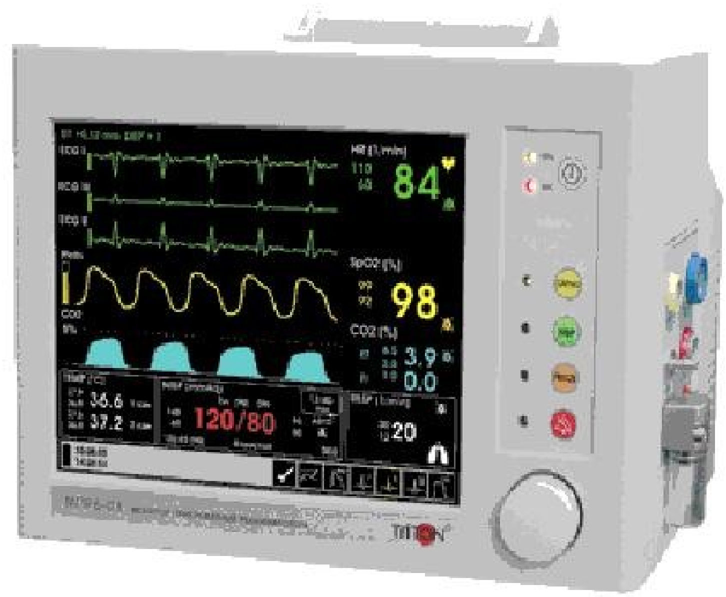


The Variables in WRSPM



Patient Monitor

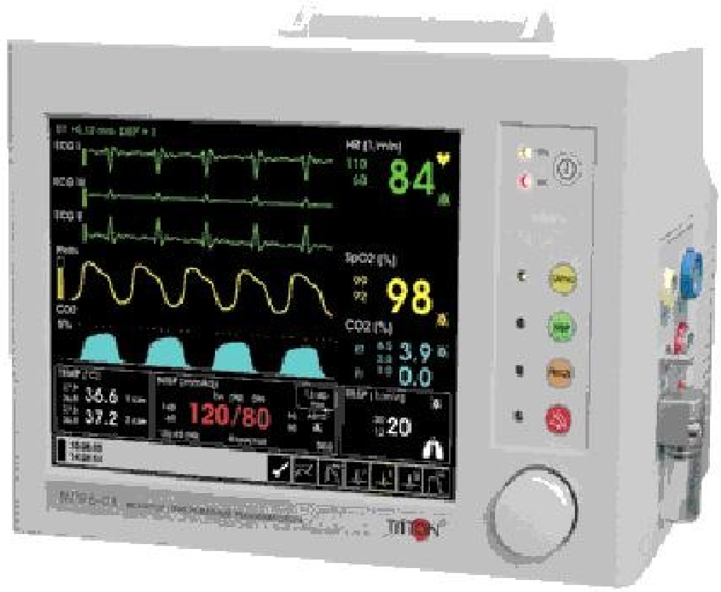
- **Desire**
  - A warning system that notifies a nurse if the patients heart stops
- **“Real” Requirement**
  - When the patient’s heart stops, a nurse shall be notified
- **“System” Requirement**
  - When the sound from the sensor (microphone taped over the heart) falls below a certain threshold, the alarm shall be actuated



For Illustration Only

Patient Monitor

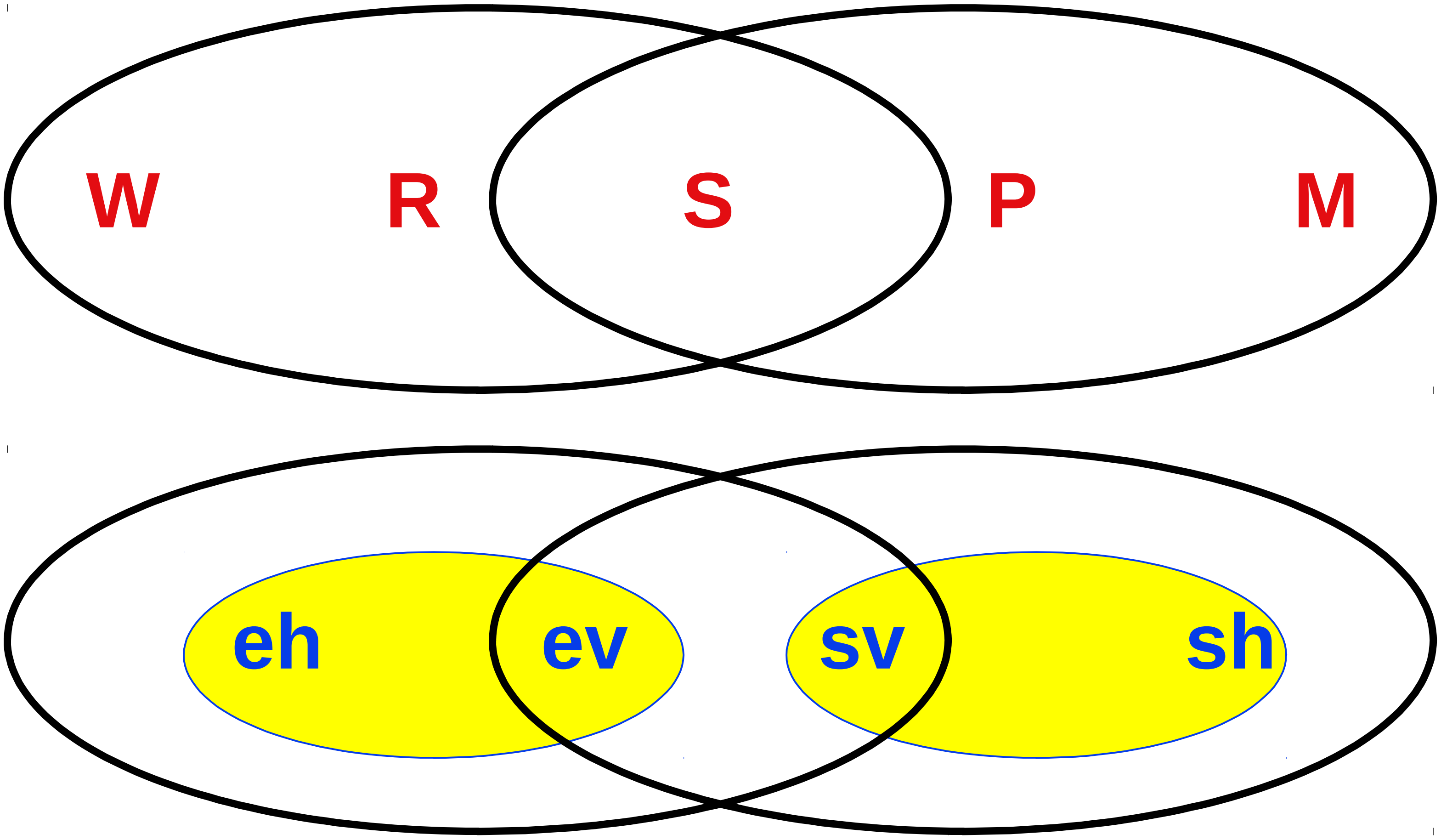
- **Desire**
  - A warning system that notifies a nurse if the patients heart stops
- **“Real” Requirement**
  - When the patient’s heart stops, a nurse shall be notified
- **“System” Requirement**
  - When the sound from the sensor (microphone taped over the heart) falls below a certain threshold, the alarm shall be actuated



For Illustration Only

Does this system satisfy the “real” requirement?

Artifacts Related to Variables



Patient Monitoring

- Requirements Definition
  - A warning system that notifies the nurse if the patients heart stops
- System Design
  - A computer that can be programmed to use a microphone as a sensor and a buzzer as an actuator
- Requirements Specification
  - If the sound from the sensor falls below a certain threshold, the buzzer shall be actuated

Patient Monitoring will Work

- If we take a computer that can be programmed to use a microphone as a sensor and a buzzer as an actuator,
- and if we program this computer to sound the buzzer when the sound from the sensor falls below a certain threshold,
- we will have a warning system that notifies the nurse if the patients heart stops

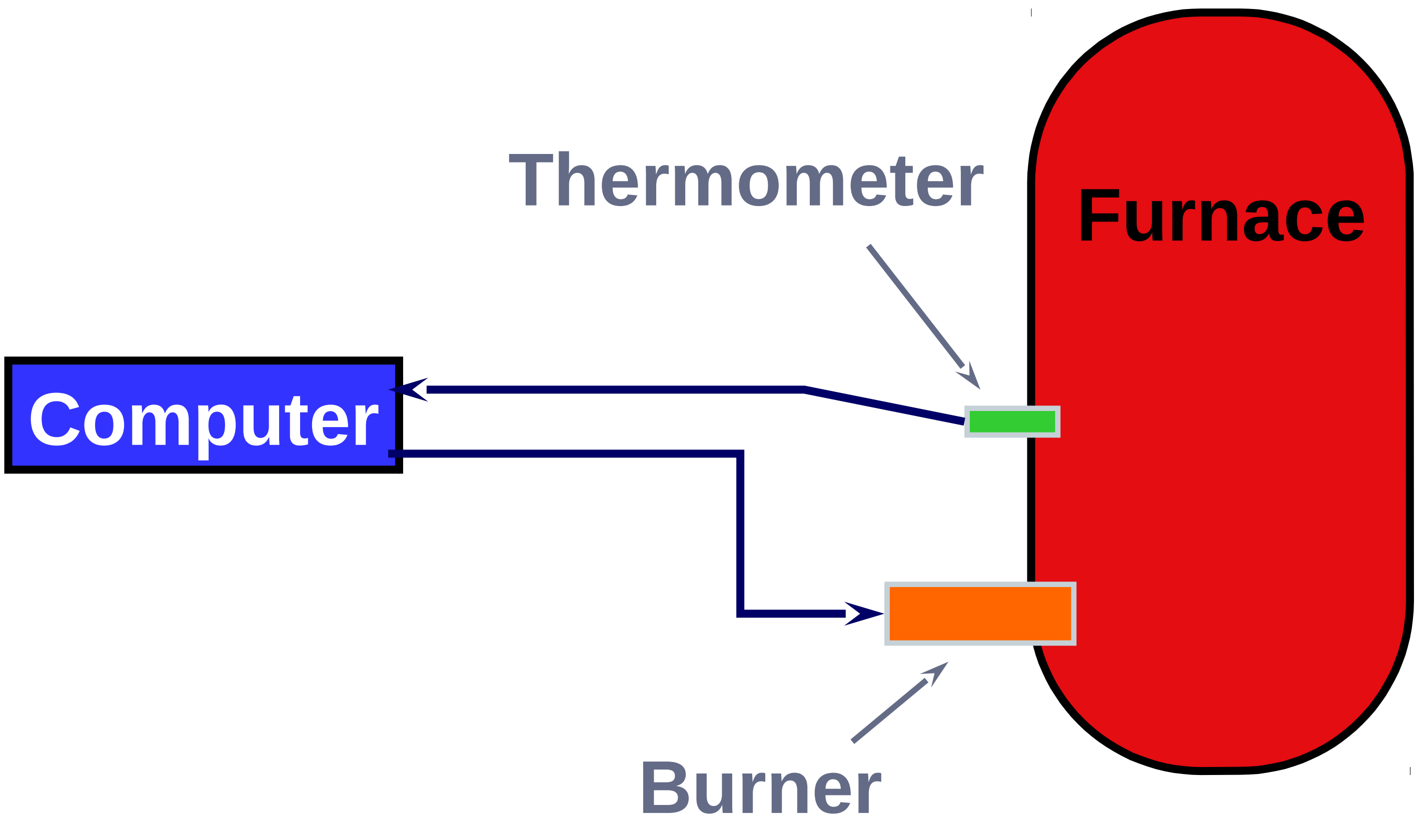
- **Do we believe this?**

Patient Monitoring will Work

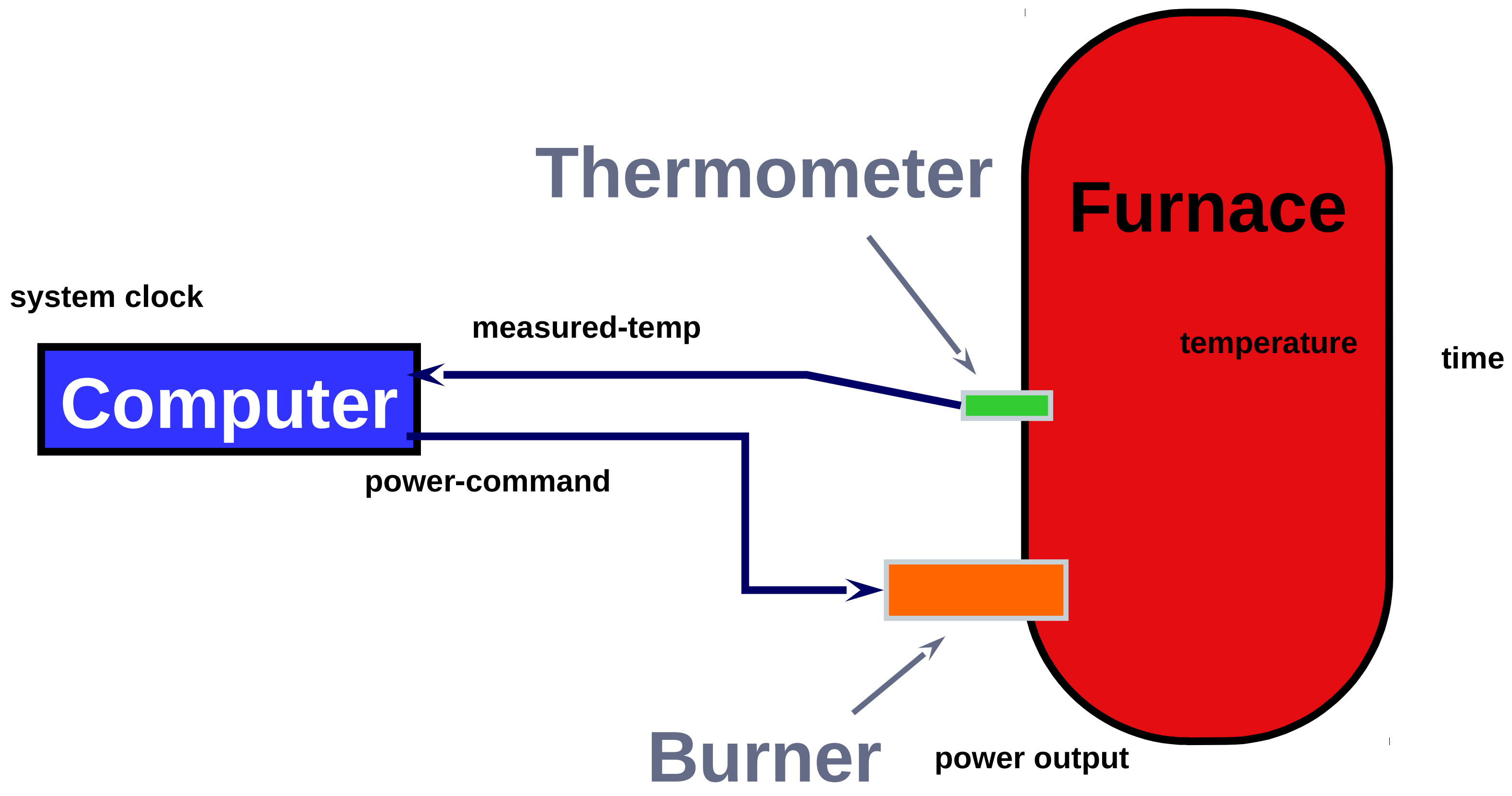
- If we take a computer that can be programmed to use a microphone as a sensor and a buzzer as an actuator,
- and if we program this computer to sound the buzzer when the sound from the sensor falls below a certain threshold,
- we will have a warning system that notifies the nurse if the patients heart stops

- **Because**
- There will always be a nurse close enough to hear the buzzer, and
- the sound from the heart falling below a certain threshold indicates that heart has (is about) to stop

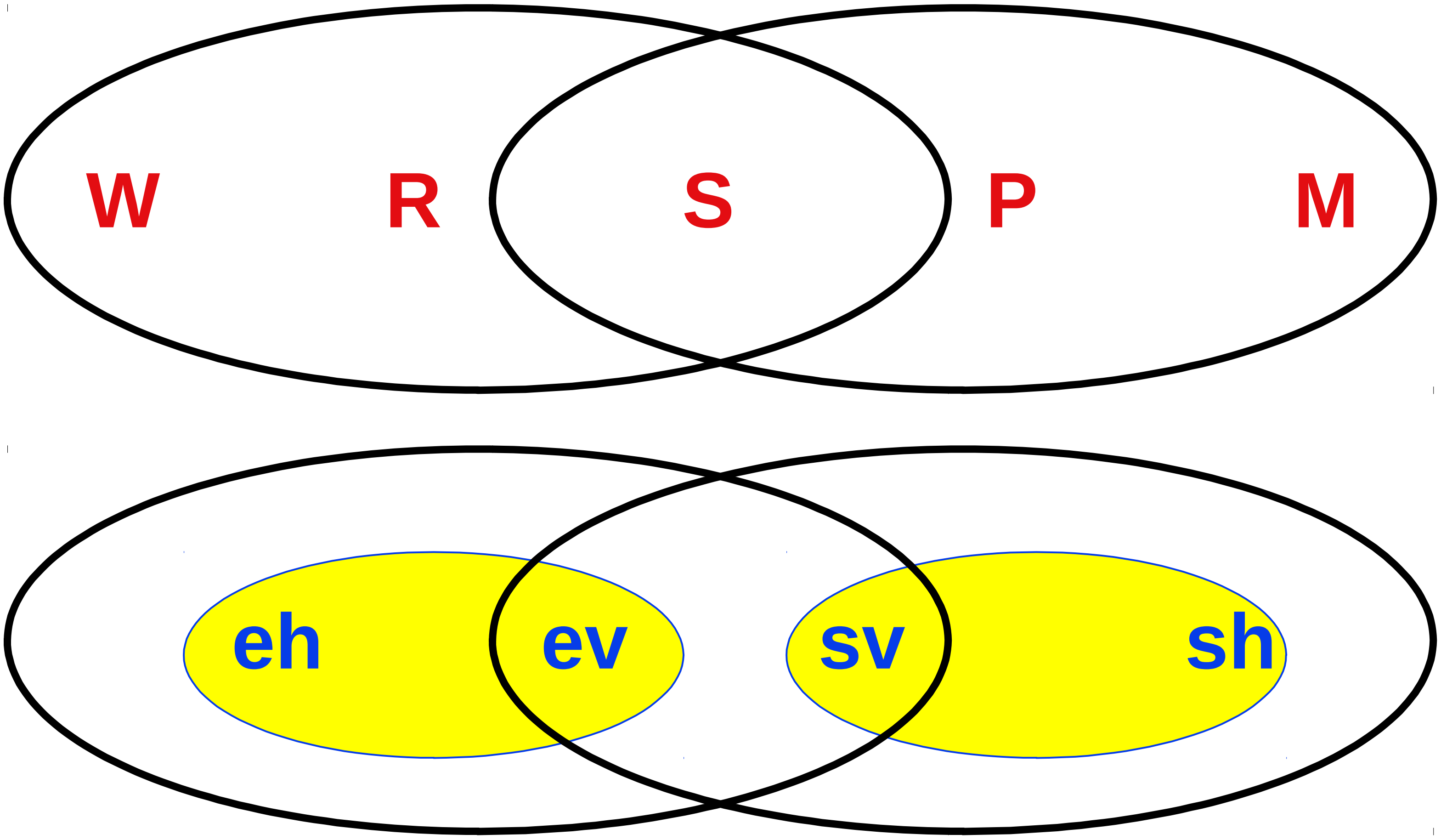
eh, ev, sv, and sh???



eh, ev, sv, and sh???



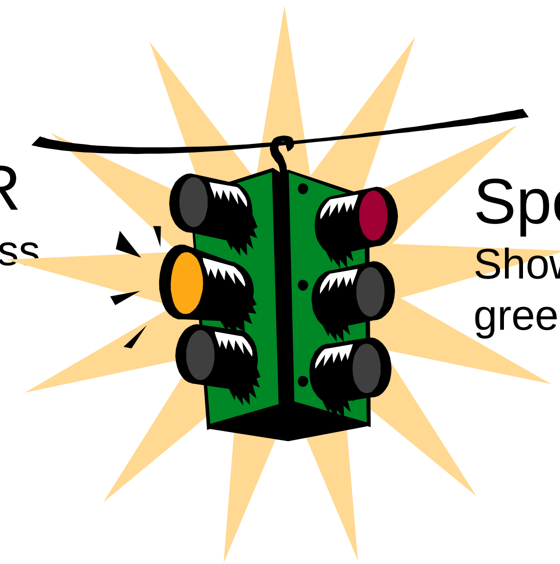
Artifacts Related to Variables



Example

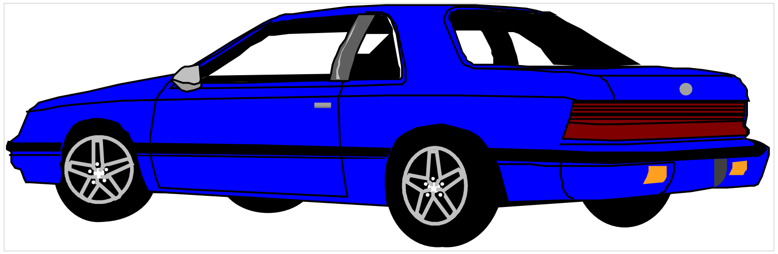
Requirement—R

Allow pedestrians to cross the road safely



Specification—S

Show a red light to the cars and a green light to the pedestrians

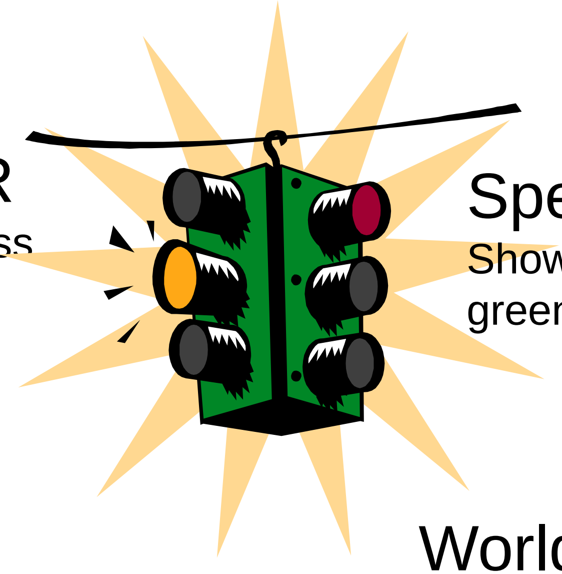


What is W so that W and S togther satisfy R?

Example

Requirement—R

Allow pedestrians to cross the road safely

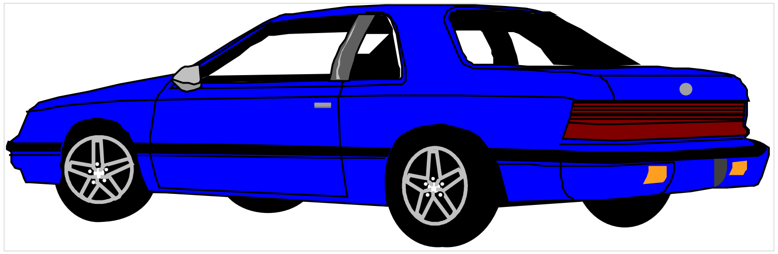


Specification—S

Show a red light to the cars and a green light to the pedestrians

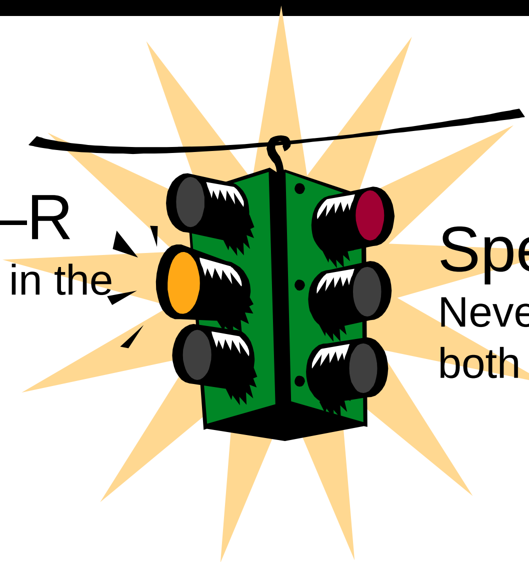
World Knowledge—W

1. Drivers stop at red lights
2. Pedestrians walk when green



W and S satisfies R

Example—Safety



Safety Requirement—R

Pedestrians and cars cannot be in the intersection at the same time

Specification—S

Never show a green light to both pedestrians and cars

World Knowledge—W

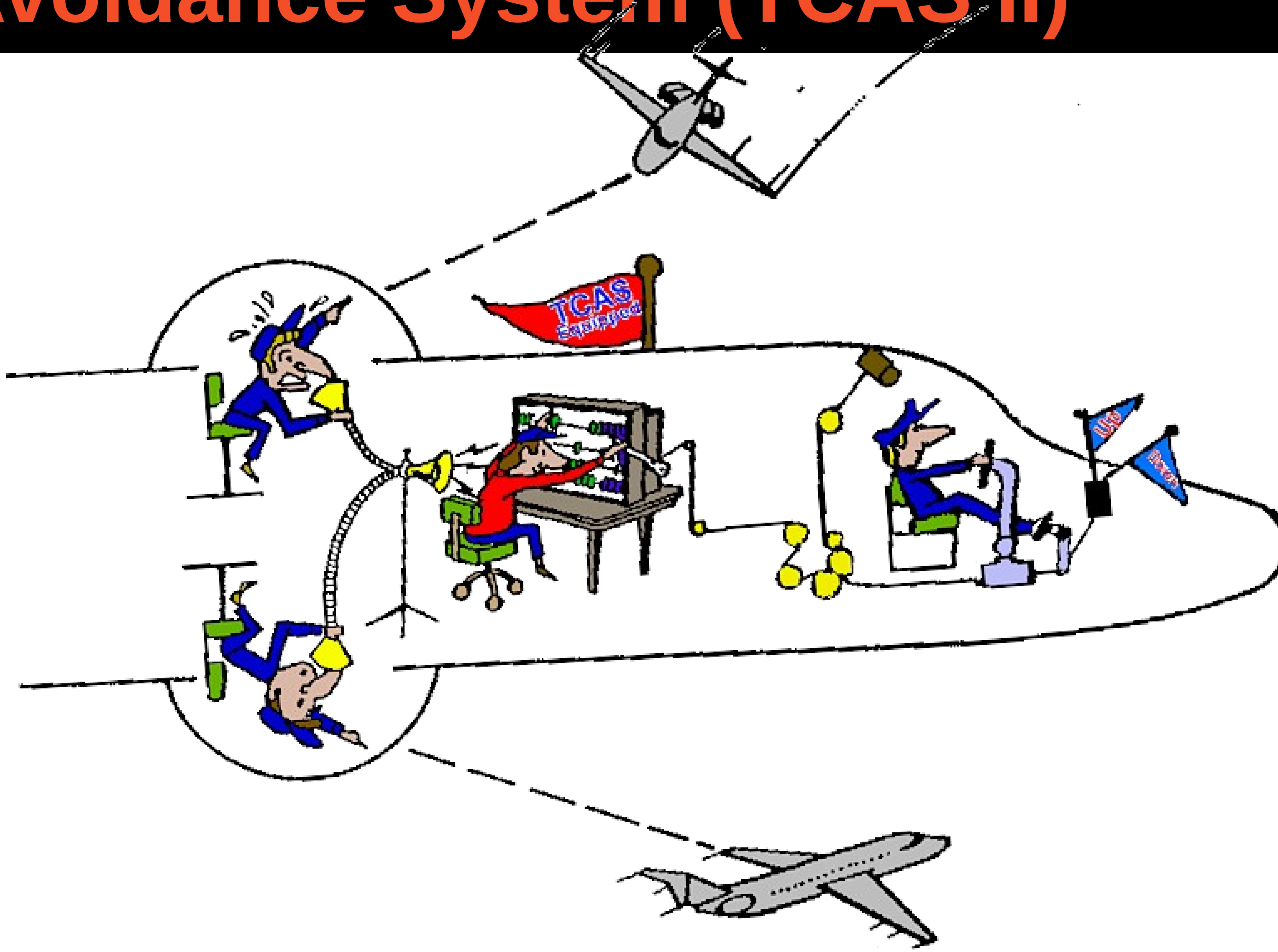
1. Drivers stop at red lights
2. Pedestrians stop at red lights
3. Drivers drive at green lights
4. Pedestrians walk when green

W and S satisfies R

World Knowledge is Essential

- This is the most error prone part of the requirements
  - Most problems can be traced to erroneous assumptions about the environment
  - Patriot missile—clock drift
  - TCAS—transponder assumptions
  - NY subway—separation not enough
- Must be rigorously validated and continually questioned

Traffic alert and Collision Avoidance System (TCAS II)



In General We Want to Show

- The specification satisfies the requirements
  - W and S satisfies R (W, S $\Rightarrow$ R)
- The implementation satisfies the requirements
  - W, M, P $\Rightarrow$ R

This is the essence of any argument that your system is "right"

- The implementation satisfies the specification
  - M, P $\Rightarrow$ S

We Have Learned

- What requirements really are
- The relationship between system and environment
  - The WRSPM model

Deriving a solution which satisfies the software requirements

Software Design
Fundamentals of Design

Today’s Objectives

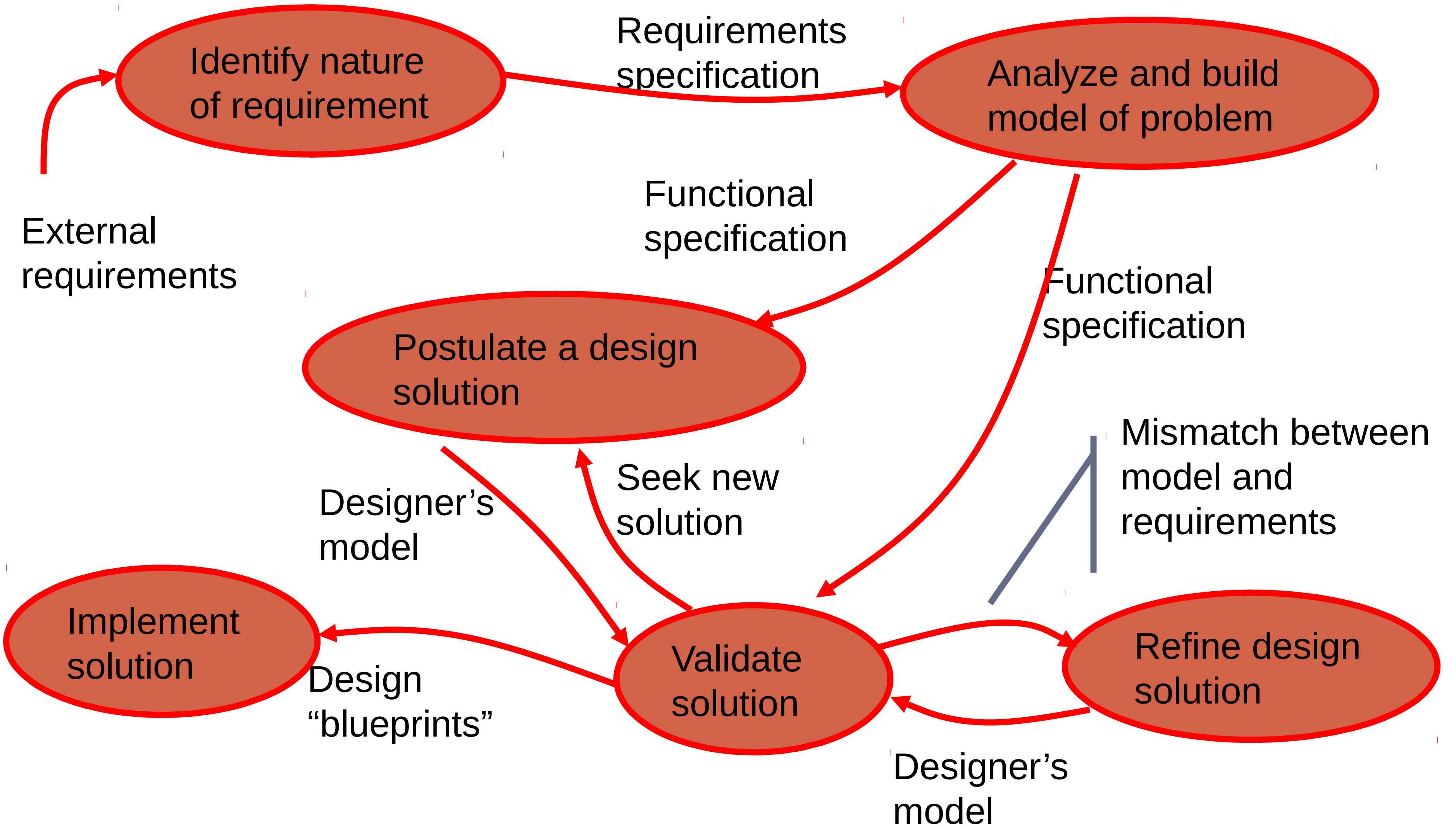
- To define design
- To introduce the design process
- To preview two design strategies
  - Functional decomposition
  - Object Oriented design
- Quick overview of design criteria

What is Design?

- Design
  - The creative process of transforming a problem into a solution
  - In our case, transforming a requirements specification into a detailed description of the software

- Design
  - The description of the solution
  - In our case, we will develop a software design

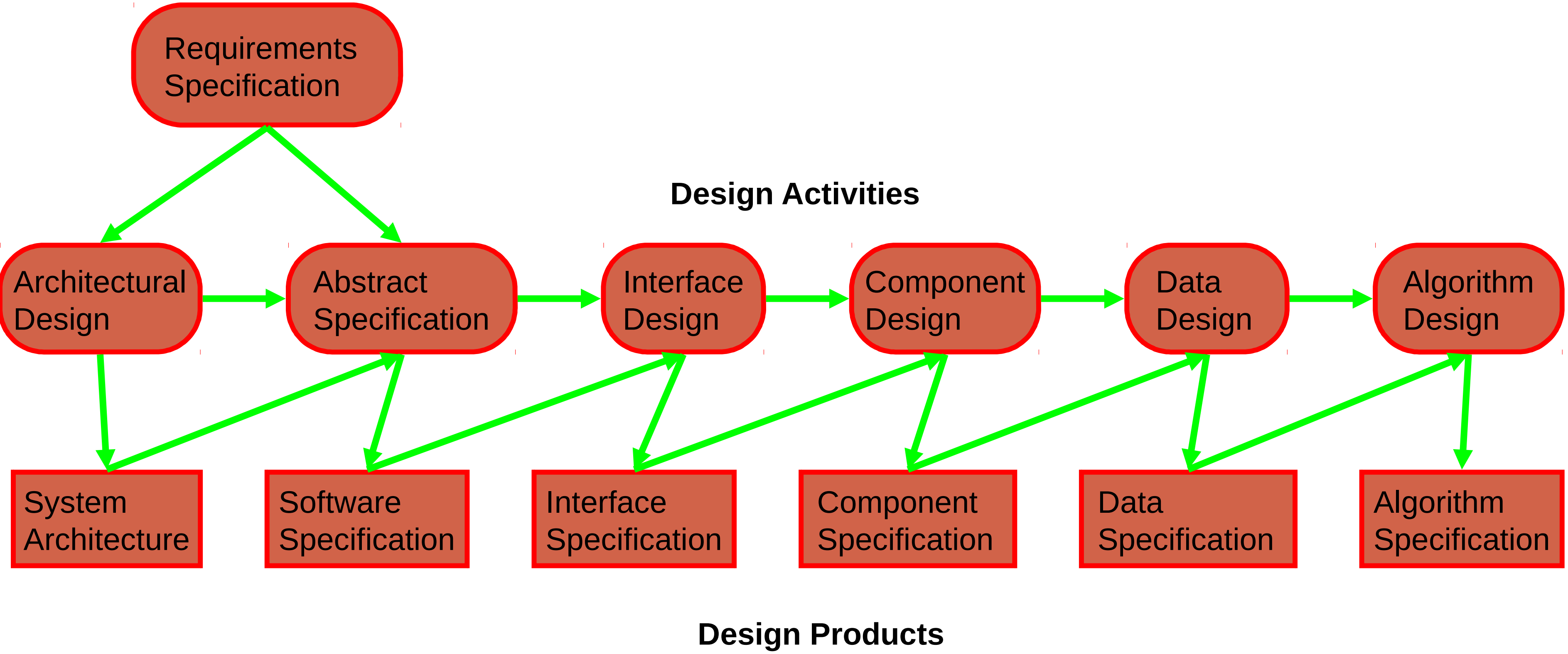
General Design Process



Stages of Design

- Problem understanding
  - Look at the problem from different angles to discover the design requirements
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design
- Repeat process for each identified abstraction until the design is expressed in primitive terms

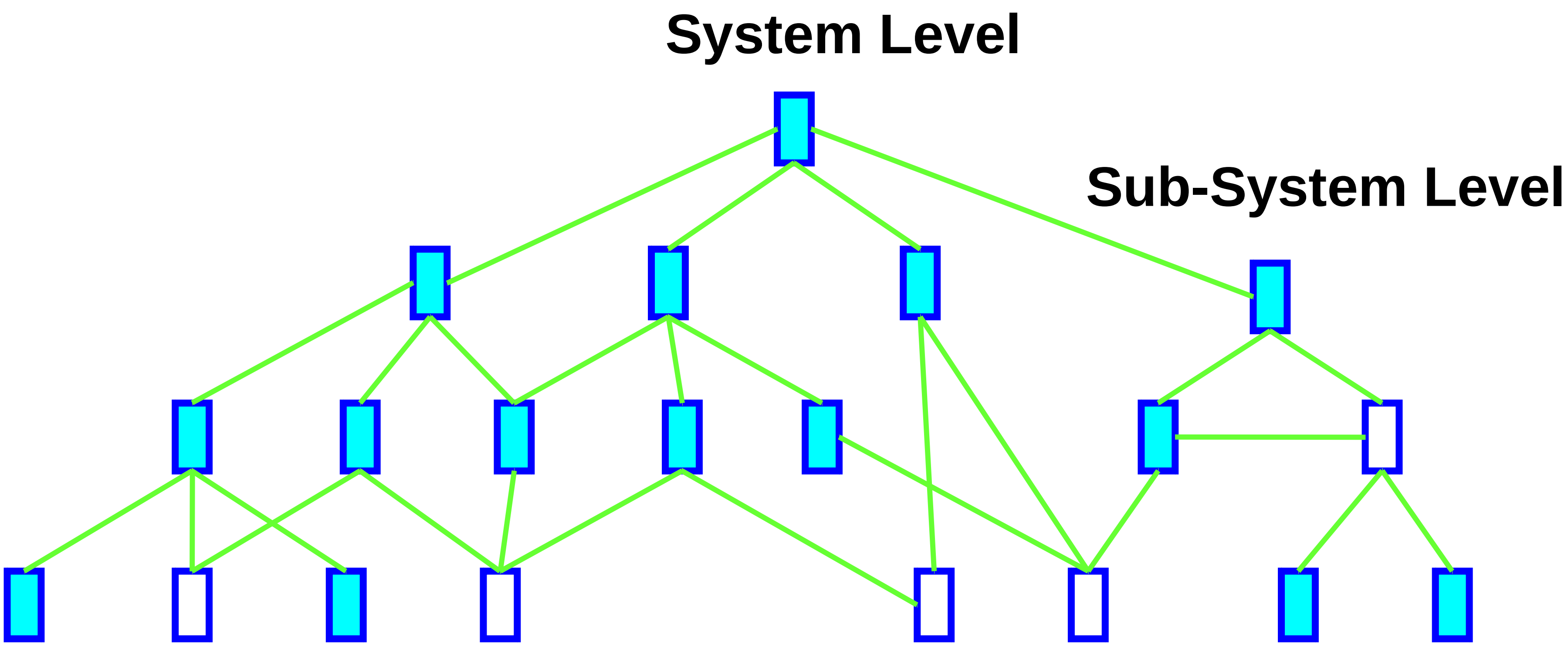
Phases in the Design Process



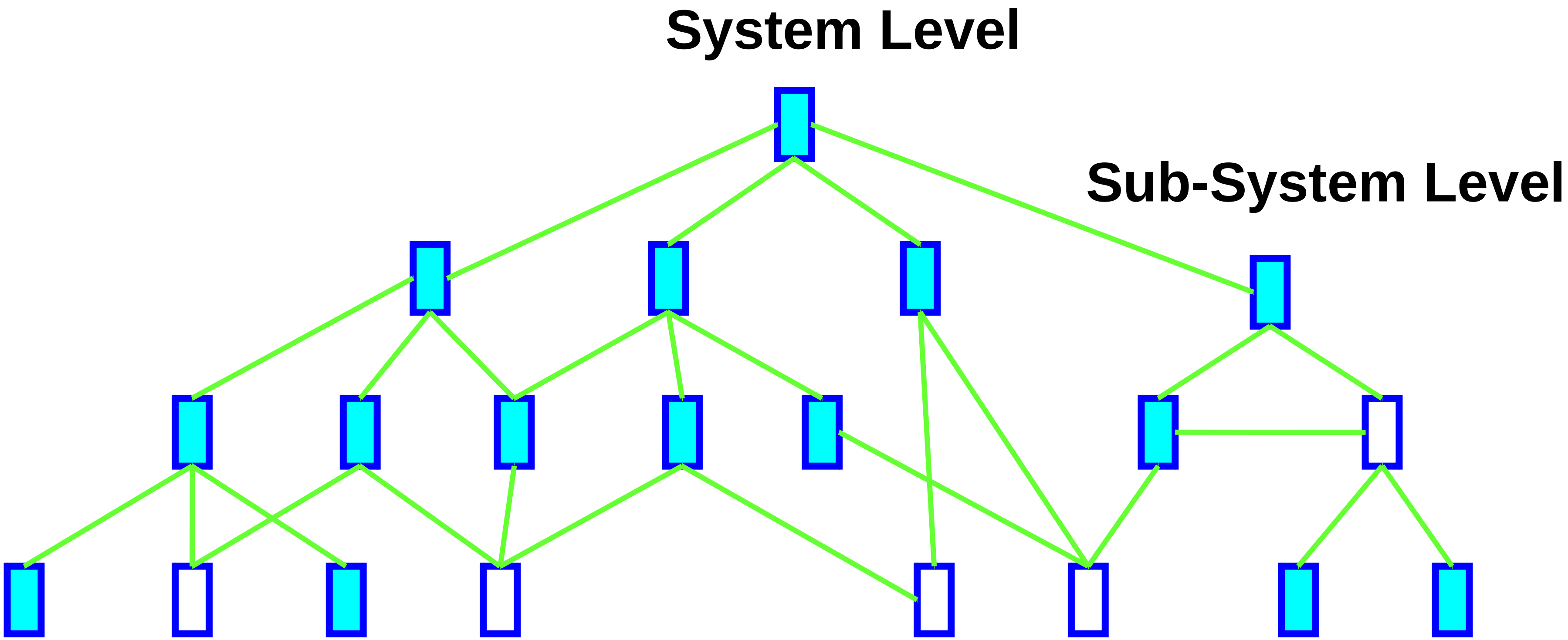
Design Phases

- Architectural design
  - Identify sub-systems
- Abstract specification
  - Specify sub-systems
- Interface design
  - Describe sub-system interfaces
- Component design
  - Decompose sub-systems into components
- Data structure design
  - Design data structures to hold problem data
- Algorithm design
  - Design algorithms for problem functions

Hierarchical Design Structure



Hierarchical Design Structure



Top-down Design

- In principle, top-down design involves starting at the uppermost components in the hierarchy and working down the hierarchy level by level
- In practice, large systems design is never truly top-down
  - Some branches are designed before others
  - Designers reuse experience (and sometimes components) during the design process

Design Description

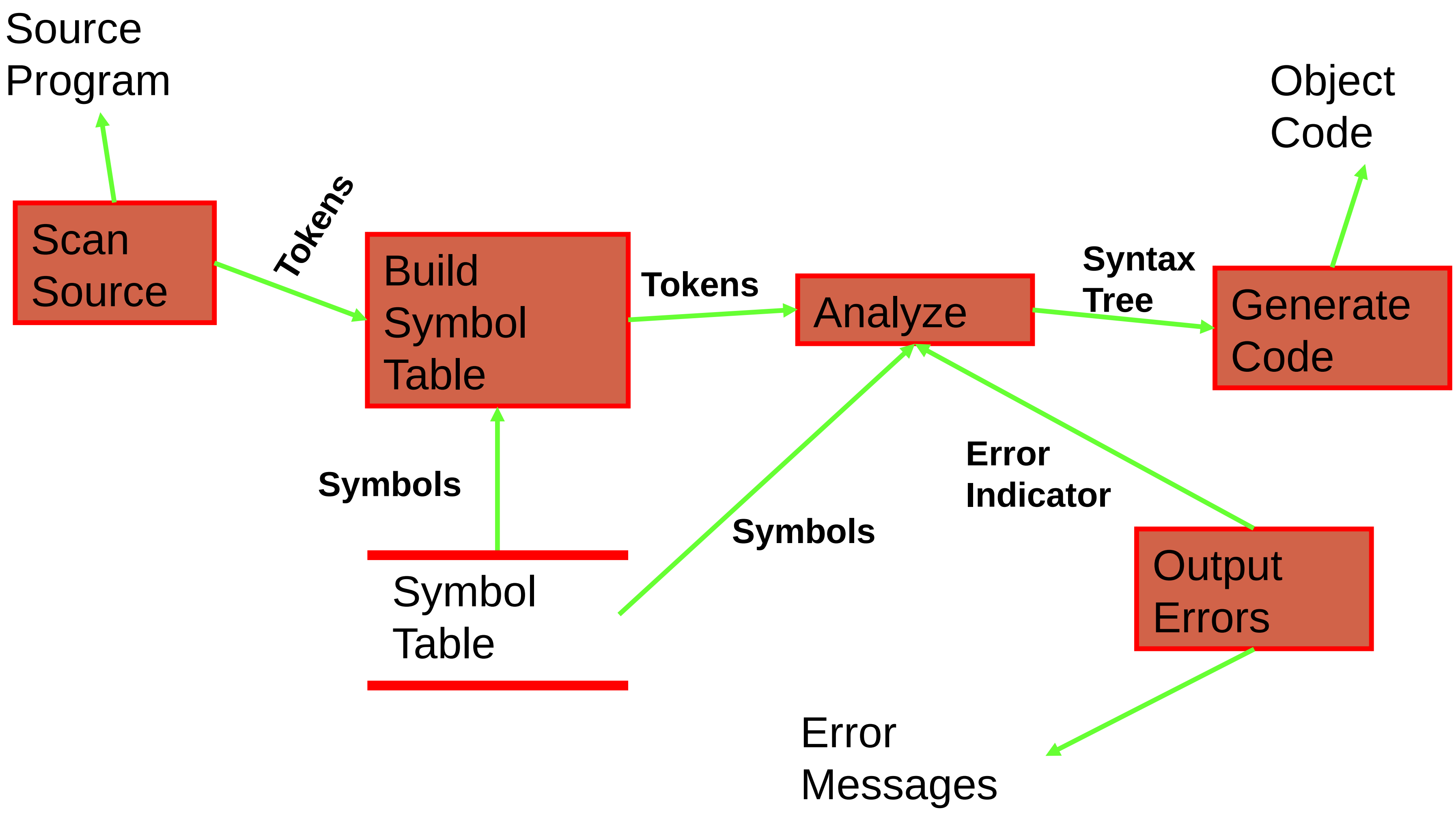
- Graphical notations
  - Used to display component relationships
- Program description languages
  - Based on programming languages but with more flexibility to represent abstract concepts
- Informal text
  - Natural language description

- All of these notations may be used in large systems design

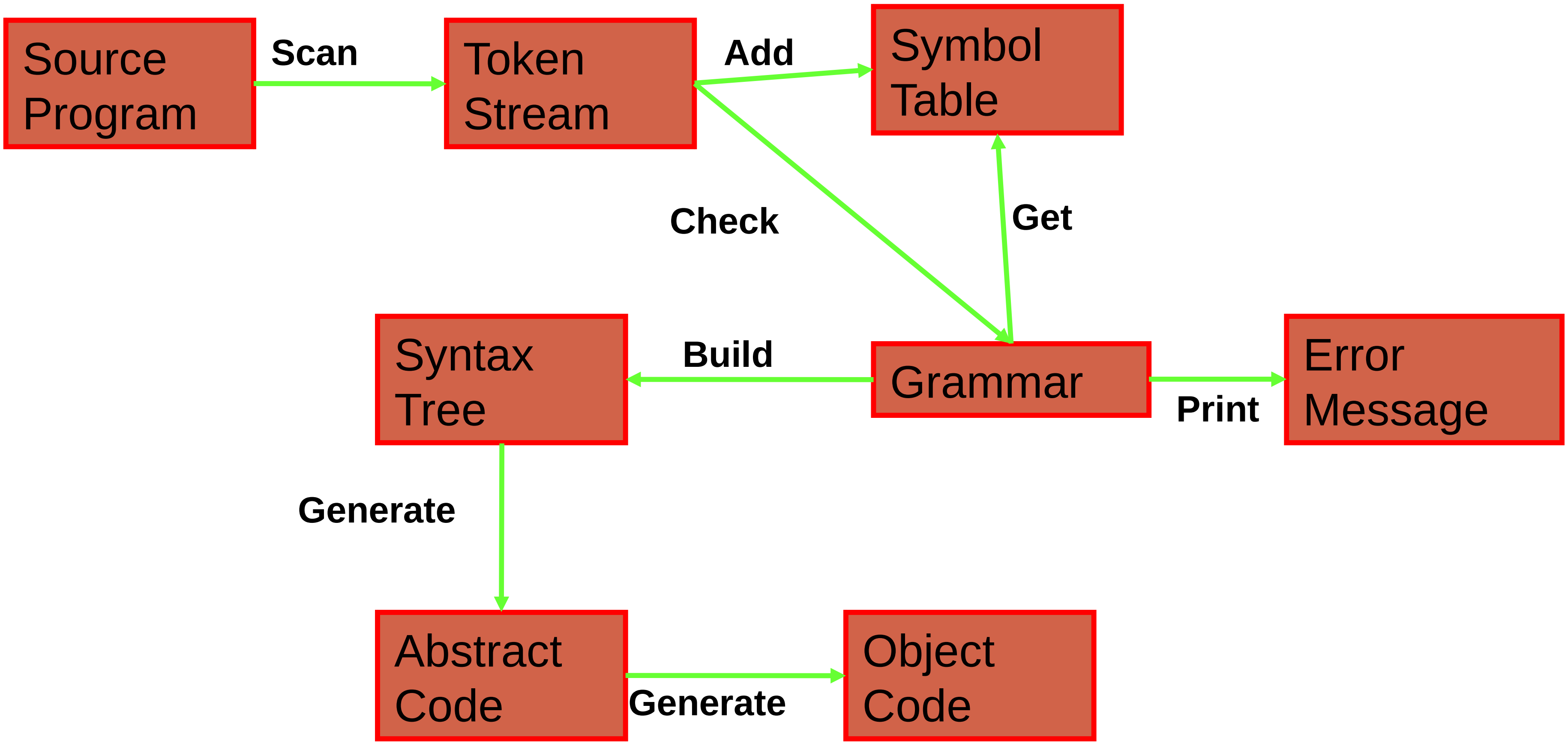
Design Strategies

- 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

Functional View of a Compiler



Object-Oriented View of a Compiler



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

Criteria for a good design

Software Design

Courtesy Mats Heimdahl

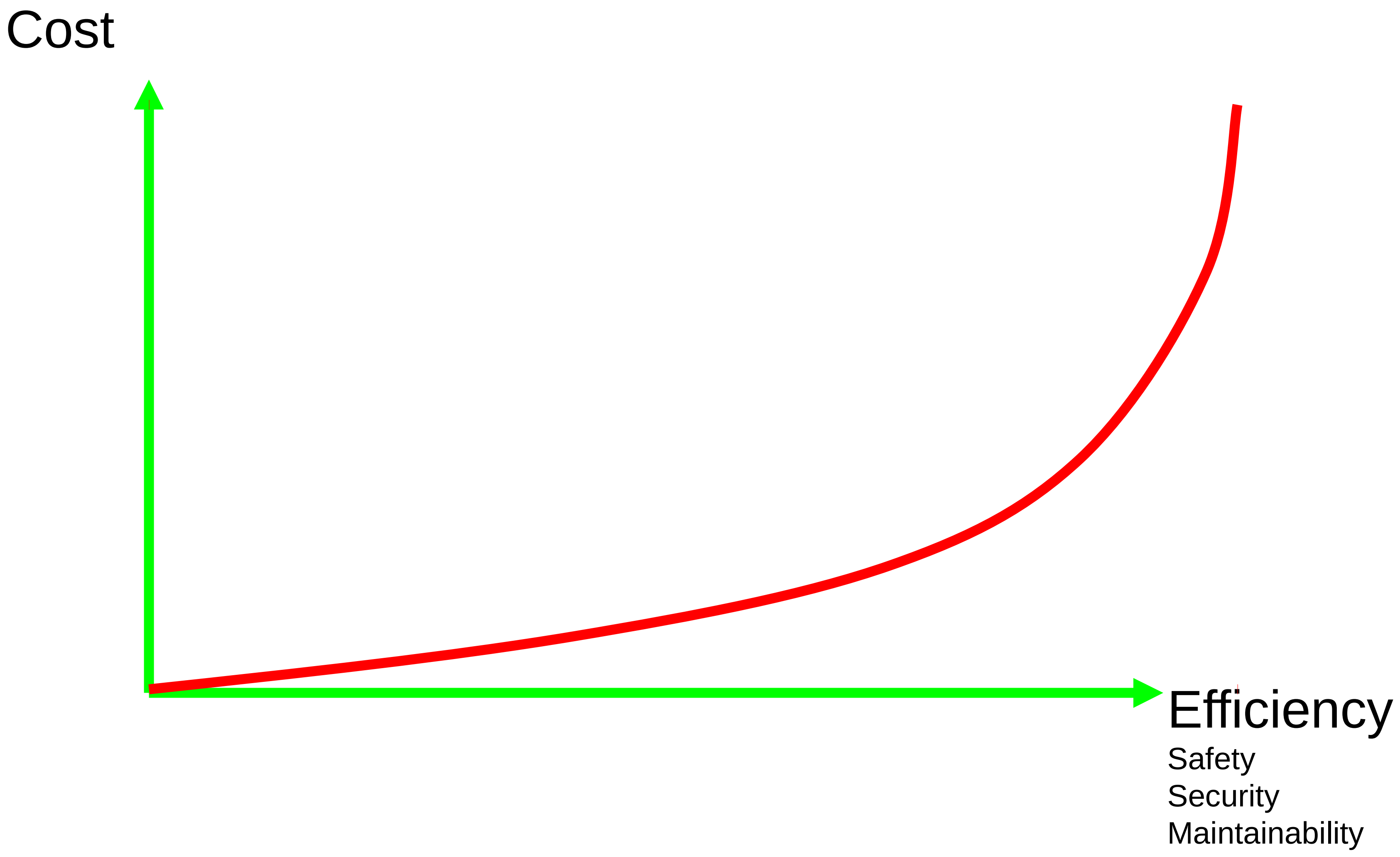
Objectives

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

Design Quality

- Design quality is an elusive concept
  - Quality depends on specific organizational priorities
- A “good” design may be the most efficient, the cheapest, the most maintainable, the most reliable, etc
- The attributes discussed here are concerned with the clarity and maintainability of the design
- Quality characteristics are equally applicable to function-oriented and object-oriented designs

Efficiency Costs



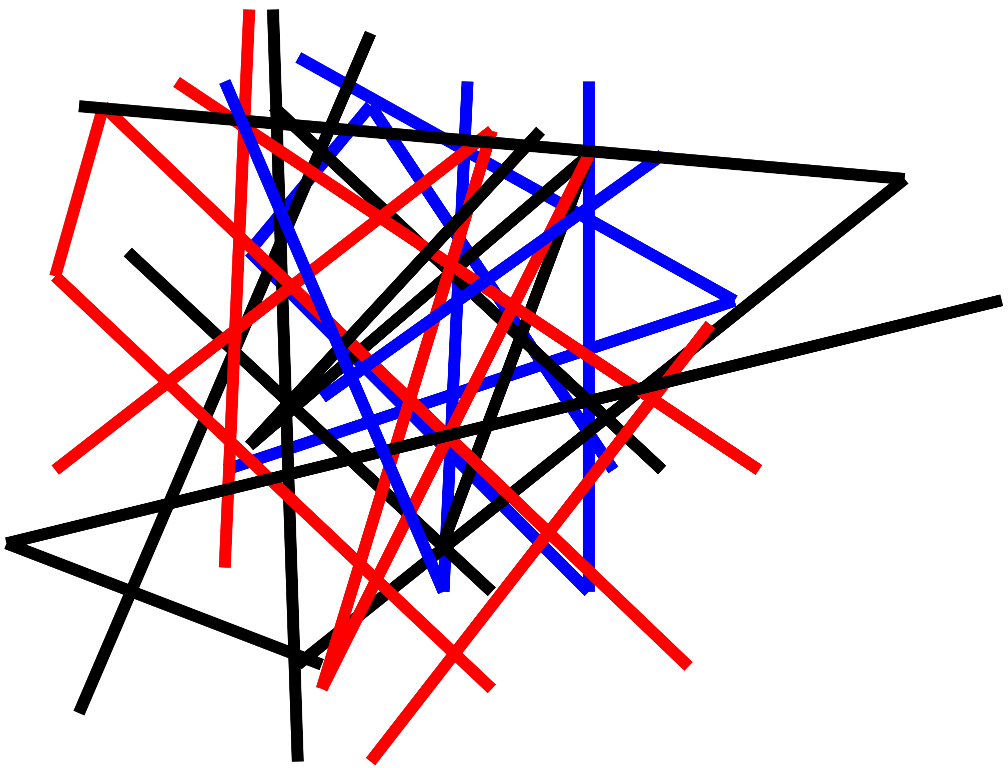
Our Focus is Clarity and Ease of Change

- Simplicity
- Modularity
  - Coupling
  - Cohesion
  - Information hiding
  - Data encapsulation
- Some “ilities”
  - Adaptability
  - Traceability
  - Etc.

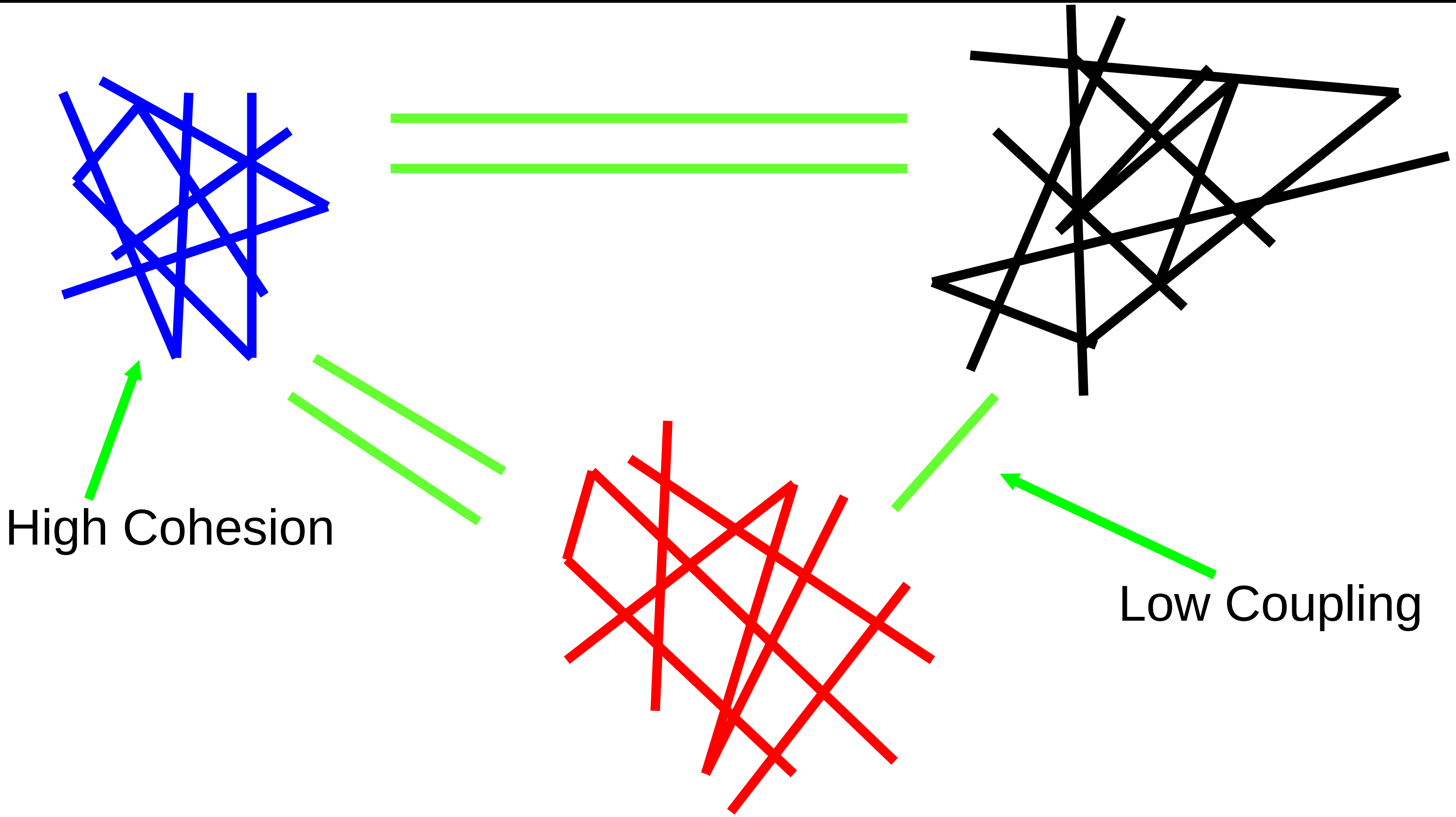
Modularity

- A complex system must be broken down into smaller modules
- Three goals with modularity
  - Decomposability
    - Break the system down into understandable modules
    - Divide and conquer
  - Composability
    - Construct a system from smaller pieces
    - Reuse, ease of maintenance, OO frameworks
  - Ease of understanding
    - The system will be changed; we must understand it
    - Understand in pieces versus understanding the whole

More Modularity



Two Essential Properties



Cohesion

- A measure of how well a component “fits together”
- A component should implement a single logical entity or function
- Cohesion is a desirable design component attribute as when a change has to be made, it is localized in a single cohesive component
- Various levels of cohesion have been identified

Cohesion Levels

- Coincidental cohesion (weak)
  - Parts of a component are simply bundled together
- Logical association (weak)
  - Components which perform similar functions are grouped
- Temporal cohesion (weak)
  - Components which are activated at the same time are grouped
- Procedural cohesion (weak)
  - The elements in a component make up a single control sequence

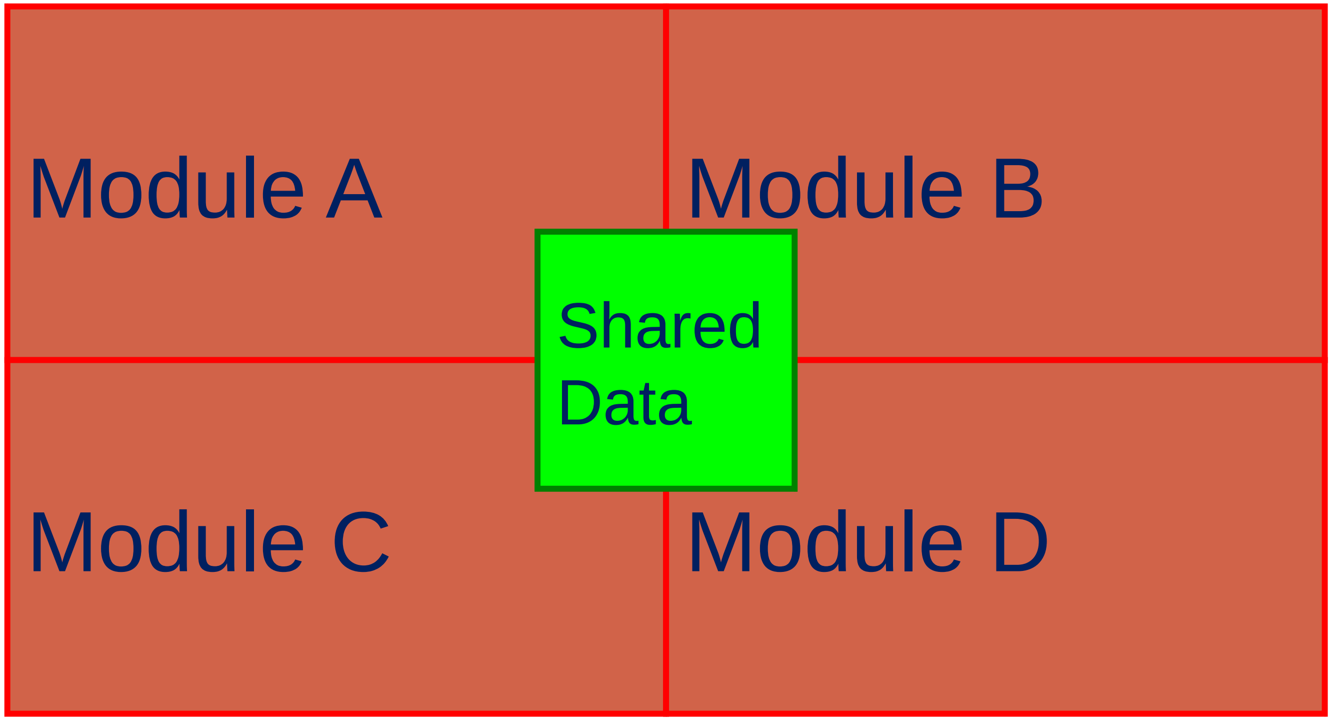
Cohesion Levels

- Communicational cohesion (medium)
  - All the elements of a component operate on the same input or produce the same output
- Sequential cohesion (medium)
  - The output for one part of a component is the input to another part
- Functional cohesion (strong)
  - Each part of a component is necessary for the execution of a single function
- Object cohesion (strong) (Data cohesion)
  - Each operation provides functionality which allows object attributes to be modified or inspected

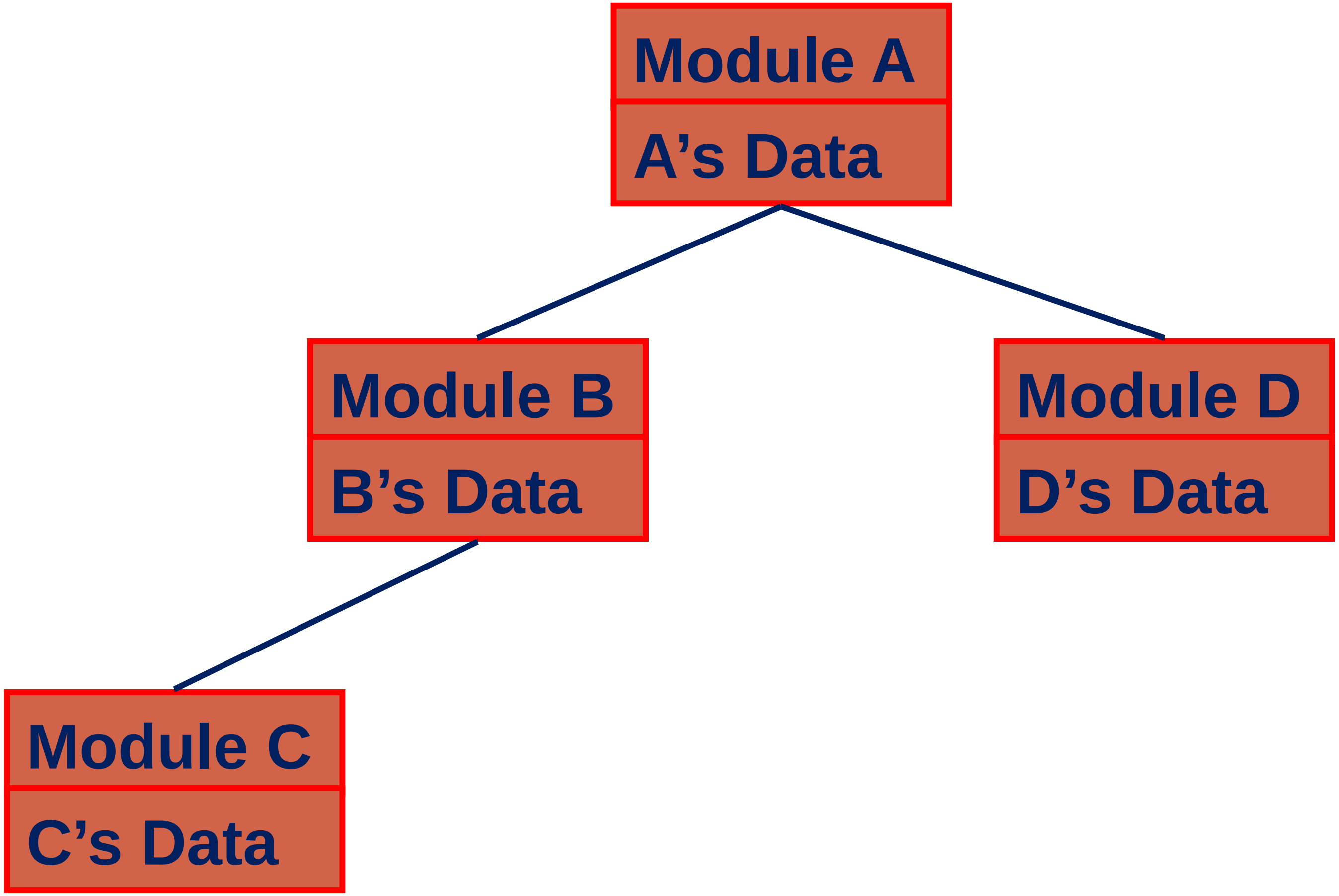
Coupling

- A measure of the strength of the inter-connections between system components
- Loose coupling means component changes are unlikely to affect other components
- Shared variables or control information exchange lead to tight coupling
- Loose coupling can be achieved by state decentralization (as in objects) and component communication via parameters or message passing

Tight Coupling



Loose Coupling



Food For Thought

- How do global variables affect coupling?
- How about large data structures?
- Classes provide a nice encapsulation mechanism and if done right provides cohesive modules
  - What does inheritance do to coupling and cohesion?

Information Hiding

- Put the complexity inside a “black box”
  - Hide it from the user of the box
  - The user does not need to know “how” the box works, just “what” it does
- Greatly reduces the amount of information the designer needs to understand at once
- Examples
  - Functions, macros, classes, libraries

Example

```
void sortAscending (int *array, int length)
```

- We do not know “what” sort routine is used
- All we need to know is what the interface is and “what” the module does

Data Encapsulation

- Encapsulate the data (or information) a module is working on
  - Protect the data from unauthorized access
  - Nobody else can mess with the data
  - If it gets corrupted, it must have been done in this module
- Helps you find where the problem is
- Makes the design more robust
  - Chances are that new additions will not mess up your old code

Example

```
int a[] ; int i, l;
void sortAscending()
{ /* body */ }


/* calling function */
a = myArray;
l = arrayLength;
i = 0;
sortAscending();
```

```
void sortAscending
(int *array, int length)
   { int i;
     /* body */ }


/* calling function */
sortAscending
(myArray, arrayLength);
```

Understandability

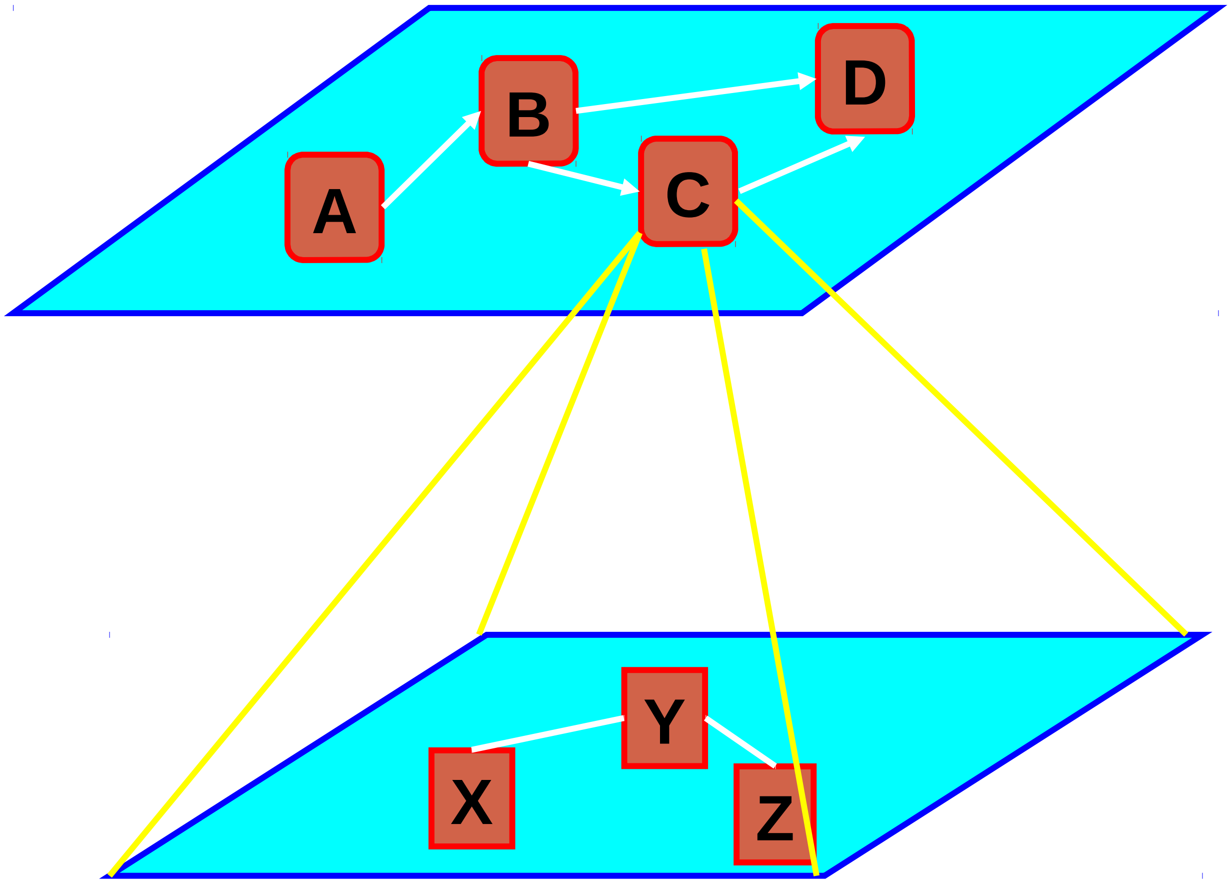
- Related to many component characteristics
  - Cohesion
  - Can the component be understood on its own?
  - Naming
  - Are meaningful names used?
  - Documentation
  - Is the design well-documented?
  - Complexity
  - Are complex algorithms used?

- Informally, high complexity means many relationships between different parts of the design
  - Hence it is hard to understand
- Most design quality metrics are oriented towards complexity measurement
  - They are of limited use

Adaptability

- A design is adaptable if:
  - Its components are loosely coupled
  - It is well-documented and the documentation is up to date
  - There is an obvious correspondence between design levels (design visibility)
  - Each component is a self-contained entity (tightly cohesive)
- To adapt a design, it must be possible to trace the links between design components so that change consequences can be analyzed

Design Traceability



Architectural Level

Functional Decomposition

Adaptability and Inheritance

- Inheritance dramatically improves adaptability
  - Components may be adapted without change by deriving a sub-class and modifying that derived class
- However, as the depth of the inheritance hierarchy increases, it becomes increasingly complex
  - It must be periodically reviewed and restructured

We Have Learned

- There are desirable design attributes
- Keep it simple!!
- Coupling and cohesion are absolutely central to good software engineering
  - Always keep this in mind!
- Information hiding and data encapsulation are almost as central
  - Always keep this in mind!