Software Architecture Specification

Gruia-Catalin Roman

October 2014

Department of Computer Science

University of New Mexico

Course Overview

1. Software Architecture Fundamentals
2. Software Architecture Specification
3. Robust Design Strategies

Software Architecture Specification

Chapter Overview

- [2.1] Documentation requirements
- [2.2] Multifaceted perspective
- [2.3] Programming abstractions
- [2.4] Modularity
- [2.5] Architectural notation
  - [2.5.1] Components
  - [2.5.2] Connectors
  - [2.5.3] Design diagrams
  - [2.5.4] Meta level specifications

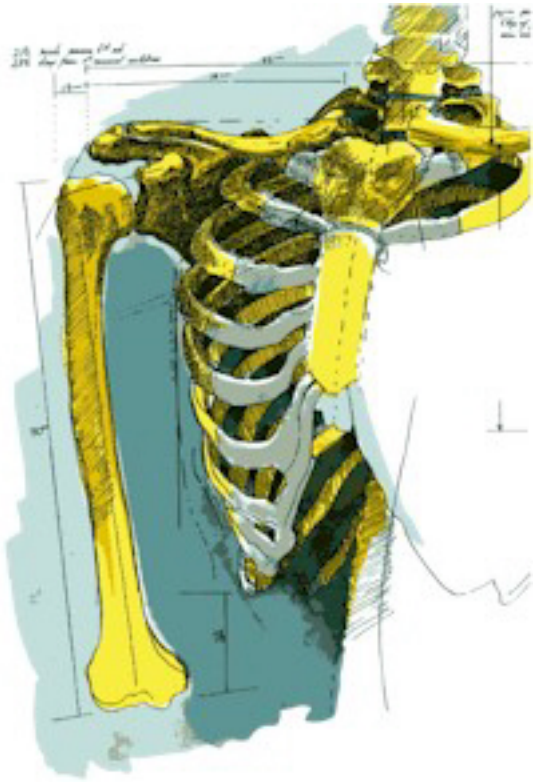
2.1 Documentation Requirements

- Key design decisions that shape the artifact we are constructing
- The rational and justification for these decisions
- Sufficient information to ensure that the final product confirms to these decisions
- Enough detail to make informed decisions about how to plan the development of the product

Documentation Strategy

Static Systems

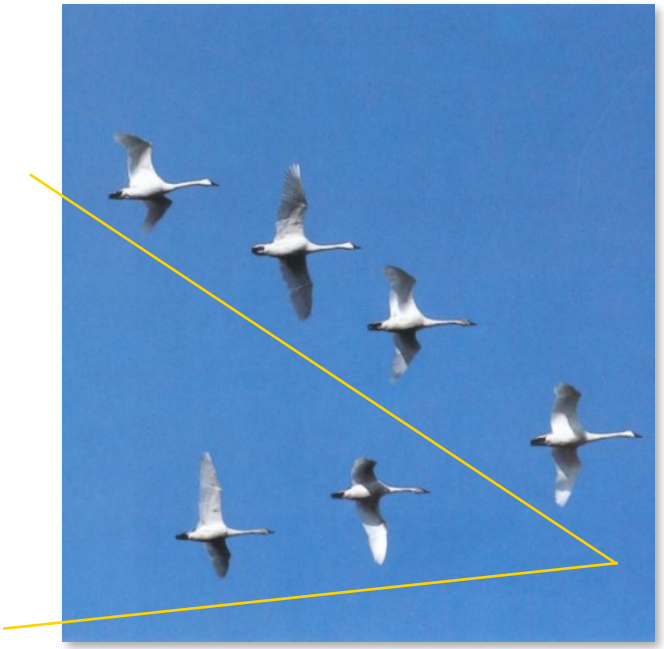
- Focus on the stable structural elements
  - system organization
- Explain local and global dynamics as overlays
  - system behavior
- Capture evaluation results
  - system analysis
- Annotate with implementation directives



Documentation Strategy

Dynamic Systems

- Focus on typical organization patterns that reveal most components and complexity
  - dynamic system organization
- Explain the evolutionary processes affecting the organization
- Highlight special and extreme cases



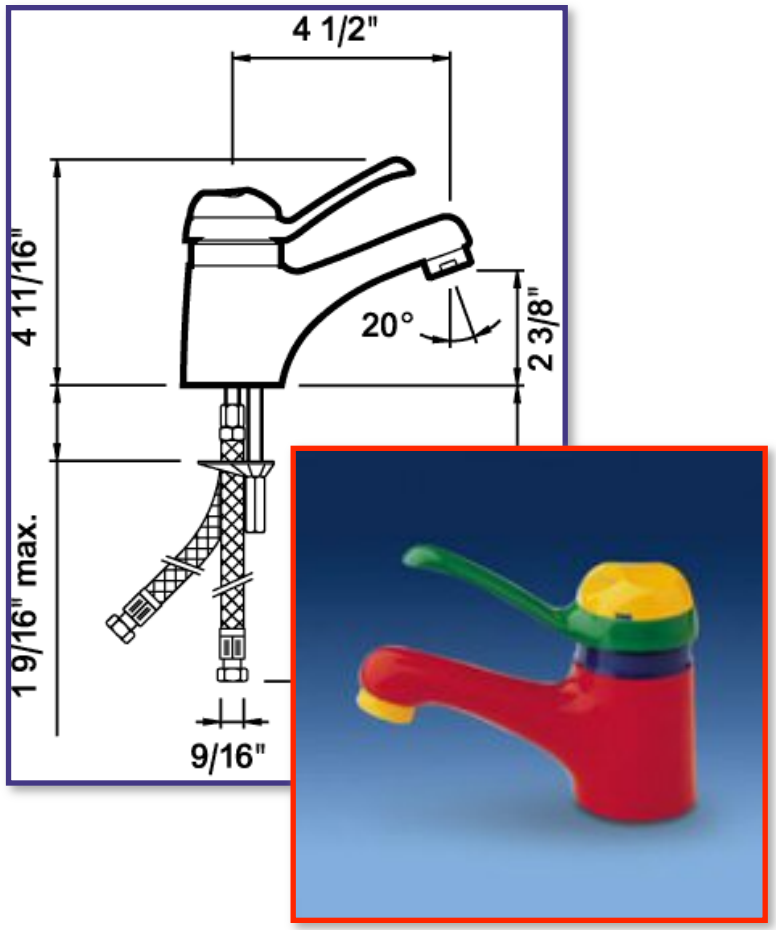
Sufficiency Criteria

An architecture is complete when all critical design issues have been addressed

- Architecture
  - as master plan
  - as management tool
  - as risk reduction program
- Modeling and analysis are critical features of a good architecture documentation
- Non-functional requirements determine the level of detail

2.2 Multiple Facets of Architecture

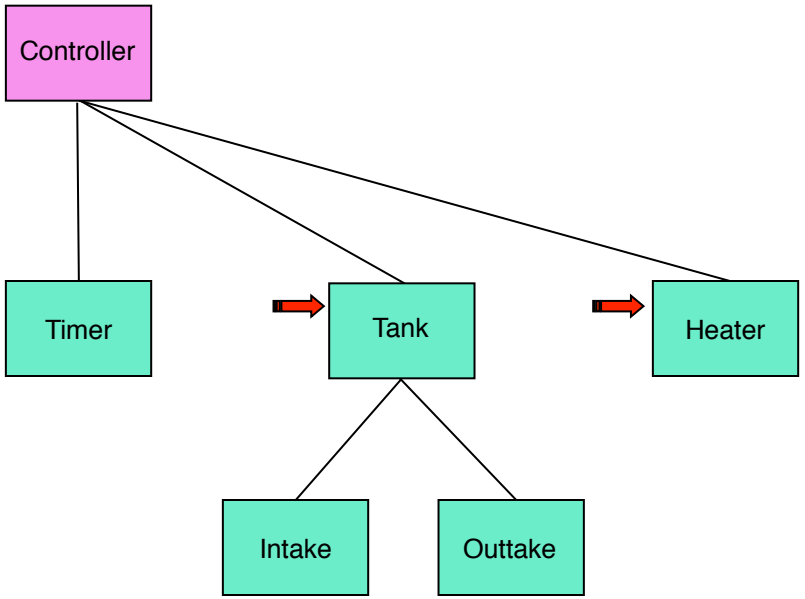
- System organization
- Component specifications
- Analytical overlays
  - behavior
  - performance
  - failure model
- Meta level specifications
- Annotations

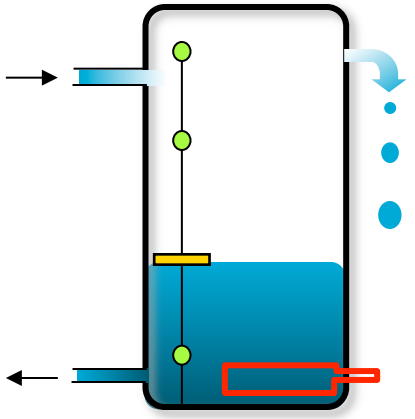


System Organization

- The core of the software architecture
- Represented as a design diagram
  - abstract view of the system structure
  - set of constraints over the system behavior
    - full behavioral specification is not feasible
- Defined in terms of
  - components making up the system
    - data, objects, threads, procedures, etc.
  - connectors constraining interactions among components
    - invocation, message passing, data access, etc.
  - actuators that trigger activity in the system
    - events, etc.
  - relations among components
    - containment, mutual dependency, coexistence, etc

Water Tank



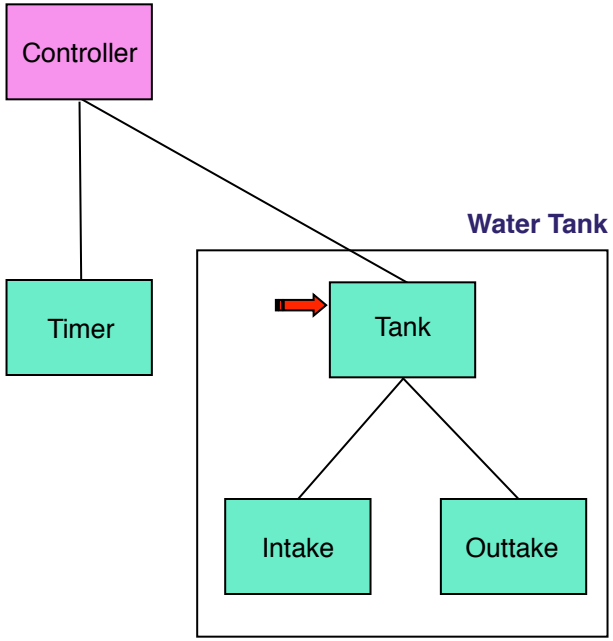


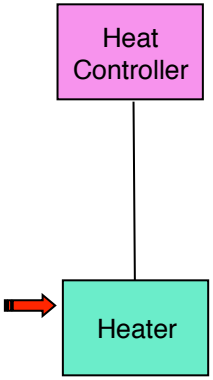
Complexity Control

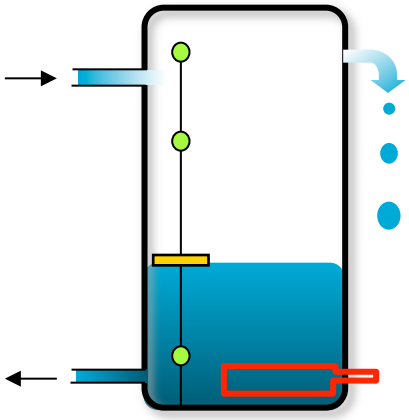
- Partition
  - separation of concerns
- Hierarchy
  - levels of abstraction
- Multiple views
  - the divide and conquer advantage
  - the price of consistency management
- Evolutionary rules
  - minimalist perspective
  - emerging structures

Water Tank

Hierarchy and partition



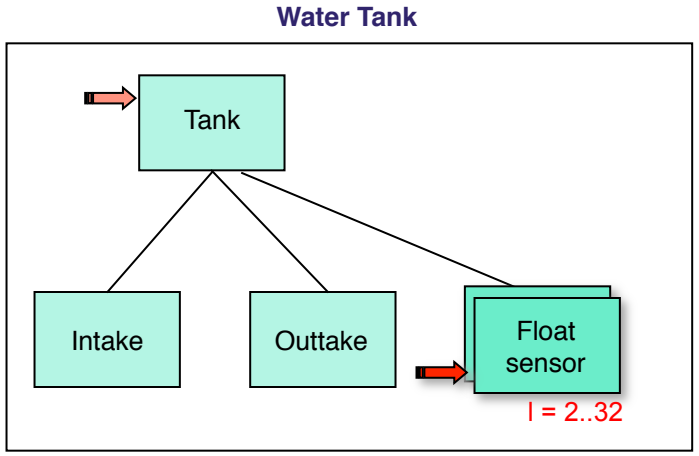


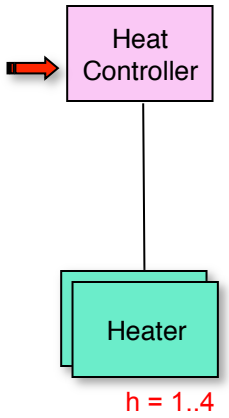


Water Tank

Configuration dynamics

- number of heating elements is $1 \leq h \leq 4$
- number of level sensors is $2 \leq l \leq 32$





Design Elements

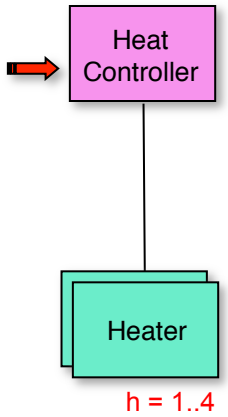
- Formally distinguished in terms of
  - components
    - software modules having direct representation in the code
  - connectors
    - abstract constructs characterizing the interactions among components
  - actuators
    - abstract mechanisms that trigger activity in the system
- Simple or composite
  - complex components may be viewed as simple at the level of architecture but composite at development time
- Characterized by
  - interface
  - behavior (state and state transitions)
- Rooted in the tradition of modular design

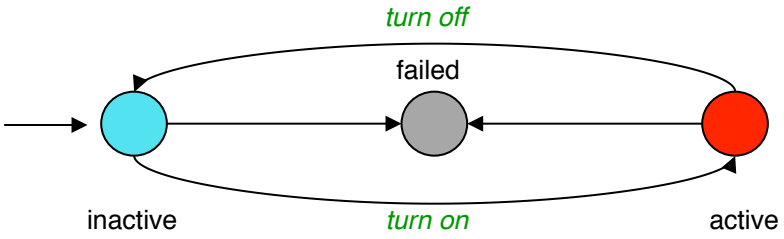
Component

- Sample component types
  - procedure
  - object
  - process (threaded object)
- Interface depends on component type and language—it is real
- Behavior reflects the component type but must be abstract

Heater

- Methods
  - turn on, turn off
  - failed, heating
- State
  - active T/F
  - failed T/F



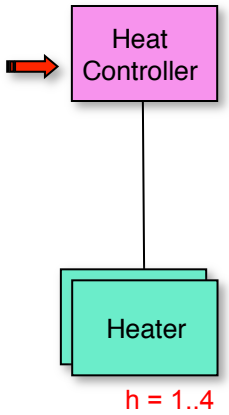


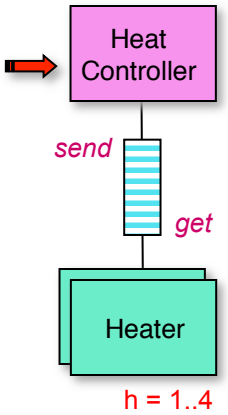
Connector

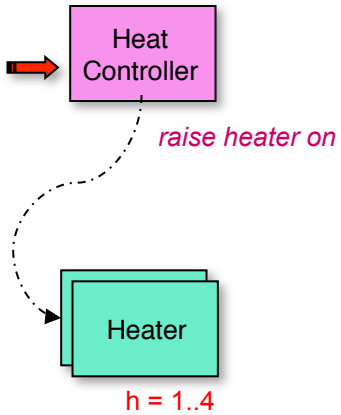
- Sample connector types
  - procedure invocation
  - message queue
- Connectors are complex specialized components

Heat Controller vs. Heater

- Possible alternatives
  - method invocation
  - message passing
  - event notification





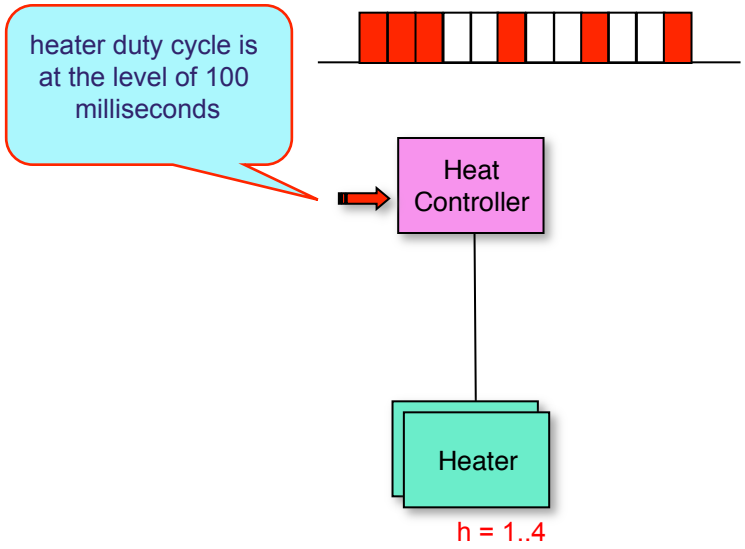


Actuator

- Actuators are events of a predefined nature that initiate activity
  - timer interrupt
  - mouse down
- Implementation is
  - language specific
  - often complex
  - distributed across the code
- Conceptually are
  - simple to understand
  - easy to analyze

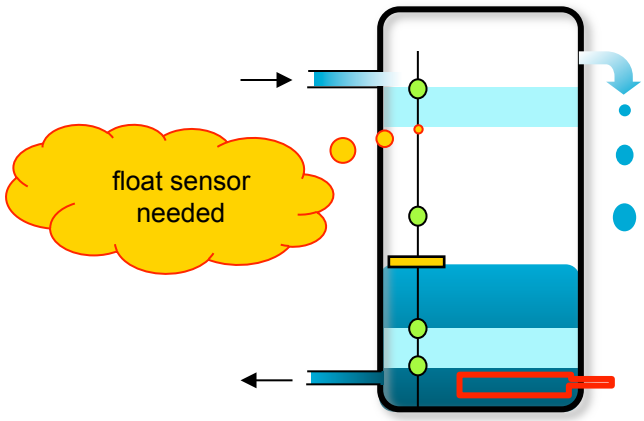
Heat Controller

- Some methods are time triggered



Analytical Overlay

- Behavior specifications not captured by the design diagram
- Analysis results and the assumptions on which they are based
- Failure models



Float sensor failure analysis

- Requirements
  - water must cover the heating element
  - water must not reach overflow outlet
- Assumptions
  - at most one float sensor fails at a time
  - sensors can be added and removed at will
  - sensors report depth when the floater is present
- Observation
  - at least four float sensors are needed

Extra Considerations

- Annotations
  - Information important in the interpretation of the architectural design
    - implementation directives
    - non-functional constraints
    - warnings
    - notions for which no notation is available
- Meta Level Specifications
  - class definitions
  - invariant properties
  - semantic constraints
  - relations
    - among classes
    - among objects
    - etc.

2.3 Programming Abstractions Revisited

- The language we speak relates to the way we think
- The way we view programming affects the kinds of systems we construct
- The history of programming languages has been a search for the “right” abstractions

Control Abstractions

Flow of control defines the order in which instructions are executed

- Sequential flow of control is built into most machines (program counter)
- Conditional jumps allow the interpreter to skip and/or repeat code segments
- if and goto statements provide a more uniform treatment by separating the condition from the flow of control transfer
- Further reductions in complexity are achieved by the shift to structured programming (if and while and other like constructs)
- exceptions provide a structured mechanism for handling error conditions (e.g., C++ throw and try/catch)

Procedural Abstractions

Procedural abstractions laid the foundation for modular design

- macro substitution offered a mechanism for naming sections of code and inserting them as needed
- subroutines (non-recursive), introduced as a memory saving device, structured the flow of control
- blocks provide an in-line structuring construct central to all modern programming languages
- procedures (recursive) encapsulate processing thus eliminating the need to look at the way they are coded (if a specification exists!)
- functions (pure) are procedures without side effects

Data Abstractions

Data is central to the representation of state

- built-in data types provide essential but primitive forms of data representation and operations on them
- programmer-defined types facilitate tailoring data selection to the specifics of the application
- strongly-typed languages improve dependability by doing strict compile-time checking
- abstract data type (ADT) is a formal characterization of a set of data structures sharing a common set of operations
- generic types (e.g., templates in C++) allow for parameterized definitions

Abstract Data Type

Provides data encapsulation and information hiding

- Defines interfaces for accessing data
- Data is accessible only through operations provided explicitly by the programmer
- Internal data representation is not visible and can change without affecting the ADT
- Creation and destruction can be automated

Abstract Specification

Unbounded stack

```
signature Stack(item)

empty() returns Stack

push(Stack, item) returns
   Stack

pop(Stack)
   returns Stack

top(Stack) returns item

is_empty(Stack)
   returns boolean
```

```
axioms

pop(push(s,x)) = s

pop(empty()) = abort

is_empty(empty())
   = true

is_empty(push(s,x))
   = false

top(push(s,x)) = x

top(empty()) = abort
```

Stack Specification in C++

```
typedef int T;
class Stack {
   public:     Stack (int size);
               ~Stack ();
               void push (const T &item);
               void pop (T &item);
               int is_empty ();
               int is_full ();
private:       int top, size;       T *list; };
```

Stack A(100), B(50);

Concurrency

Concurrency impacts the choice of components and connectors

- Concurrency is often relegated to operating systems services rather than language definition (e.g., C++)
- coroutines introduced the notion of passing the flow of control among subroutines
- concurrent process (e.g., task in Ada, thread in C++) provides for logically parallel execution
- Inter-process communication assumes a variety of forms
  - shared variables
  - message passing
  - remote procedure call
- Synchronization (mutual exclusion, barriers, etc.)

2.4 Modularity Principles

- Modularity is a complexity control method
- Modules are parts of the system
  - may be developed and tested independently
  - may be understood (usually) on their own
  - may be composed and decomposed
- Modular design is the only acceptable design practice in engineering today

Established Principles

- Separation of concerns
  - divide and conquer
- Clean abstraction
  - define simple and stable interfaces
- Information hiding
  - minimize the impact of changes
  - localize the consequence of errors

Basic Questions

- What precisely is a module?
- How should modules be selected?
- How should modules be described?
- How should modules interact?

Two Possible Answers

Function-oriented modularity

- Synonyms: action, process, transformation
- Basic idea: organize the system structure around the kinds of activities it must perform
- Rationale: a system is the functionality it supplies to the user

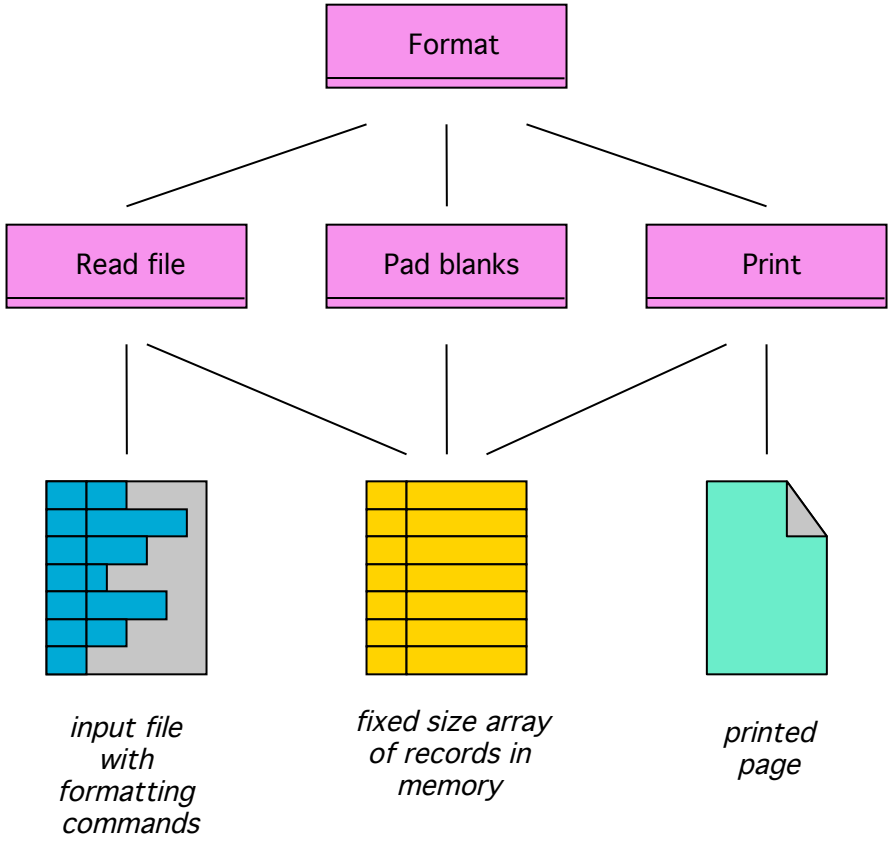
Object-oriented modularity

- Synonyms: data
- Basic idea: organize the system structure around the kinds of data it must maintain
- Rationale: functionality changes more often than the data required to support it

Function-Oriented Modularity

- Key concept:
  - procedure (procedural abstraction)
- Approach:
  - a system is organized hierarchically with procedures at one level calling procedures at the levels below
  - typically, the system is designed using top-down stepwise refinement
- Weaknesses:
  - insufficient attention is paid to data structures design
  - without encapsulation information hiding is hard to enforce
  - functionality is subject to change thus leading to design instability

Functional Decomposition



Impact of Change

- Changes often impact large sections of the code due to the lack of encapsulation of data and devices
  - the formatted text no longer fits in memory (all procedures are affected)
- Component reuse is difficult to achieve
  - the same input format is needed to implement a duplicate file command (parts of Read file need to be removed)
- Subcontracting is undermined
  - the data structures need to be redesigned (all subcontractors are affected)
- A better design could avoid these problems but the methodology does not encourage us to do it

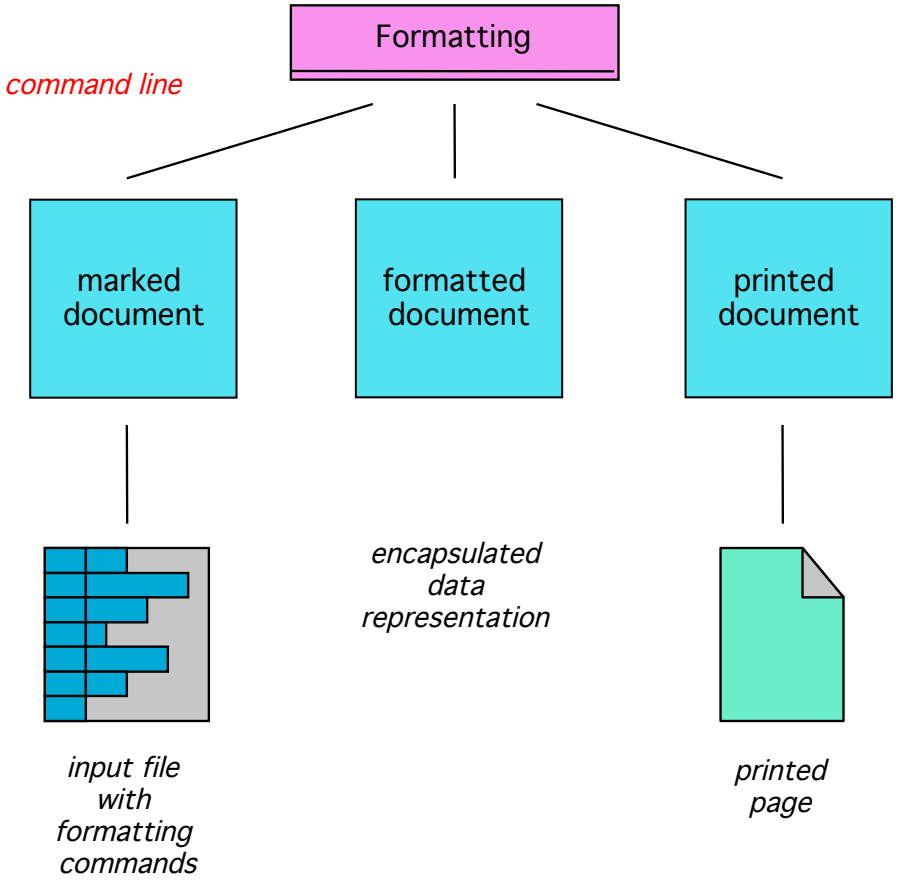
Object-oriented modularity

- Key concept:
  - abstract data type (data abstraction)
- Approach:
  - a system is organized around abstract views of the data structures it controls
  - typically, a certain degree of bottom-up design is required
- Strengths:
  - data structures are encapsulated
  - emphasis is placed on achieving generality and stability

Design Methodology Implications

- *Object-Oriented Design* is a methodology which promotes a program organization consisting of an algorithm which manipulates a set of programmer-defined objects
- The set of objects is typically static
- The software objects often correspond to objects existing in the problem domain
  - traditional data structures: integer, array, stack, queue, set, sequence, list, etc.
  - application specific data structures: radar attributes, display chain, raster image, etc.
  - interfaces to virtual devices: display, button, printer, sensor, etc.
- This form of OOD can be implemented without support from an object-oriented programming language

Object Decomposition



2.5 Architectural Notation

- Components
- Connectors and actuators
- Diagrams
- Meta-level specifications

2.5.1 Component Notation

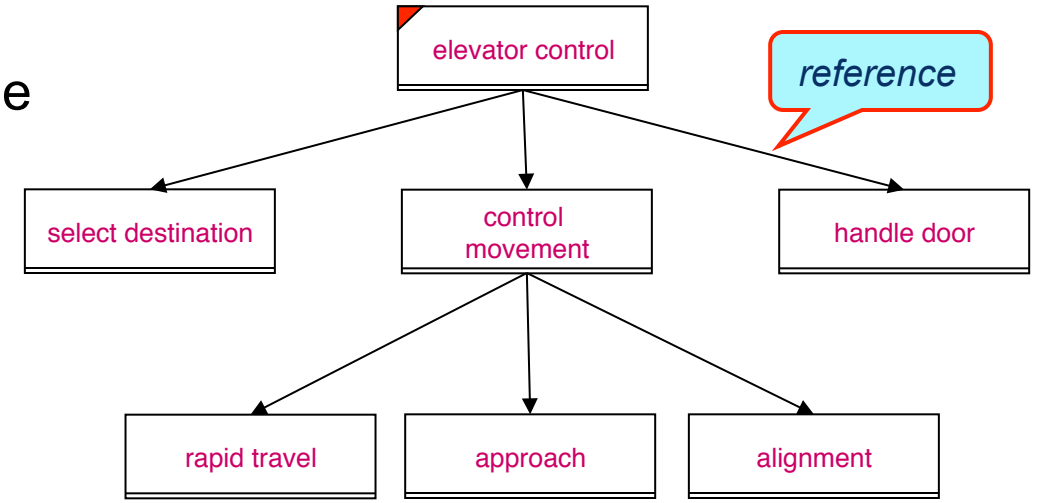
- Passive
  - procedure
  - object
- Active
  - task
  - active object
- Organizational
  - package
- External
  - devices and interfaces

Procedure

- Definition
  - encapsulates an algorithm or control process
  - executes when invoked by an active component
  - retains little if any memory of its previous execution
- Specification
  - pseudocode is recommended
  - narrative and flowcharts
- Composition
  - pure hierarchical functional composition
  - reference relation defines invocation ability

Hierarchical Processing

- Functional decomposition may be employed in order to encapsulate policy decisions and to control the complexity of
  - the processing logic
  - non-trivial methods
- The relation defining the interactions among procedures is called a reference and constraints who can invoke whom



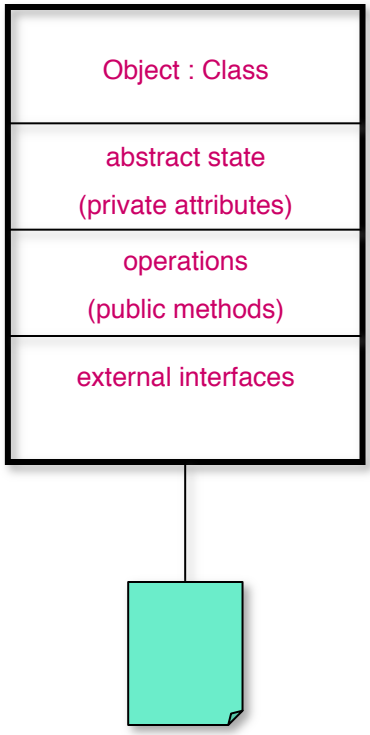
object object object object

Simple Objects

- An object is defined by the set of operations (services) it provides to its environment
- Some operations change the state of the object while others provide information about its state
- An object may be the internal abstract view of an external device and the operations that may be performed on that device
- An object may be a data structure (and its associated operations) containing information about a real-world object or some abstract concept

Notation Revisited

- A full graphical specification can capture additional details about the object
- The context in which the notation is used determines which elements are left out



Object Specification

- It is analogous to a user manual
- It must include an abstract description of the object from a user viewpoint
  - interface specification—signature
  - abstract state—private attributes
  - operations—public methods
  - functionality associated with each operation
  - encapsulated devices
  - behavior constraints (e.g., synchronization and prescribed invocation order)
- It also defines the requirements for the implementer
- It may include additional information regarding expected performance and reliability

Specification Principles

- The primary goals of an object specification are clarity and simplicity
- It must be simple and conceptually clean
- It must be presented at an abstract level
- It must be independent of any particular realization
- It must be consistent with the externally observable object behavior
- The abstract state must be clearly distinguished from data definitions which have a concrete representation in the program
- Operations must be defined in terms of the abstract state of the object

Example: Scene Manager

Let Scene S be an object that encapsulates the definition and display of a two-dimensional composition consisting of flat pictures organized in layers

Scene S: ⇐ state component, object instance

- A set of pictures initially empty.

Picture: ⇐ concept, class

- A picture is defined in terms of
  - name
  - corner coordinates
  - depth
  - pixels

Method Definition Techniques

- Narrative
- Pseudocode
- Finite state machine

Operational Specification

add_picture(P)

if a picture with the same name as P
is already part of the scene S
then **raise exception** duplicate_picture
else add P to S
endif

Axiomatic Specification

add_picture(P)

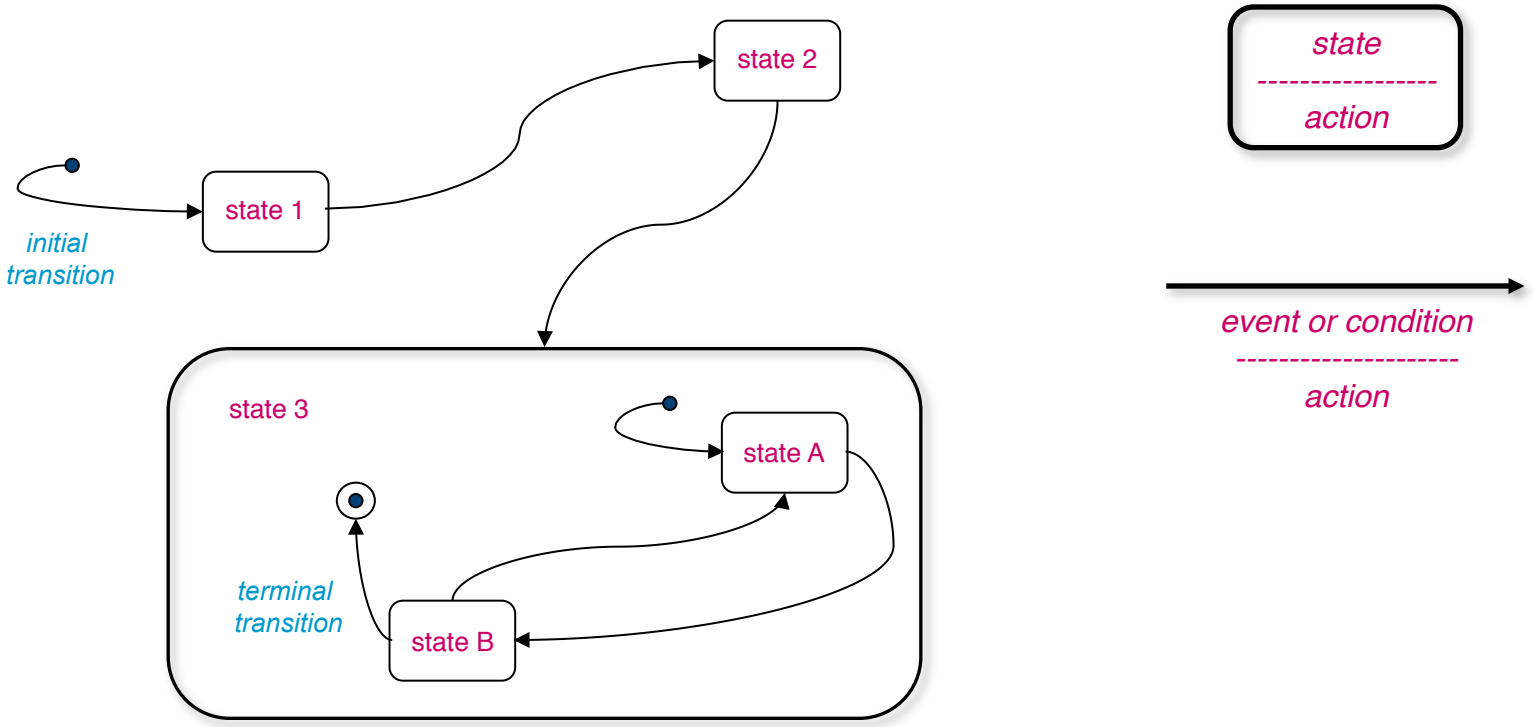
Pre: no picture with the same name as P is part of the scene S

Post: P is part of S

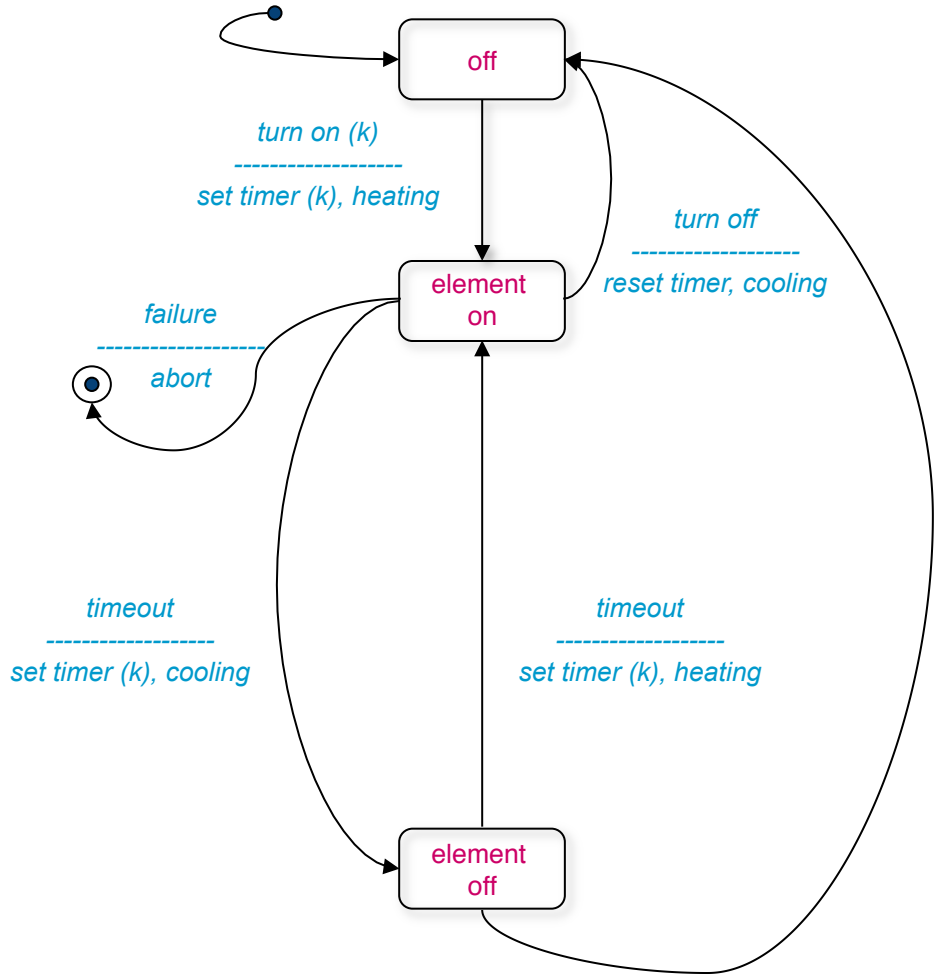
Exp: duplicate_picture

Finite State Specification

- Basic statechart notation



Example: Heating Element



Design

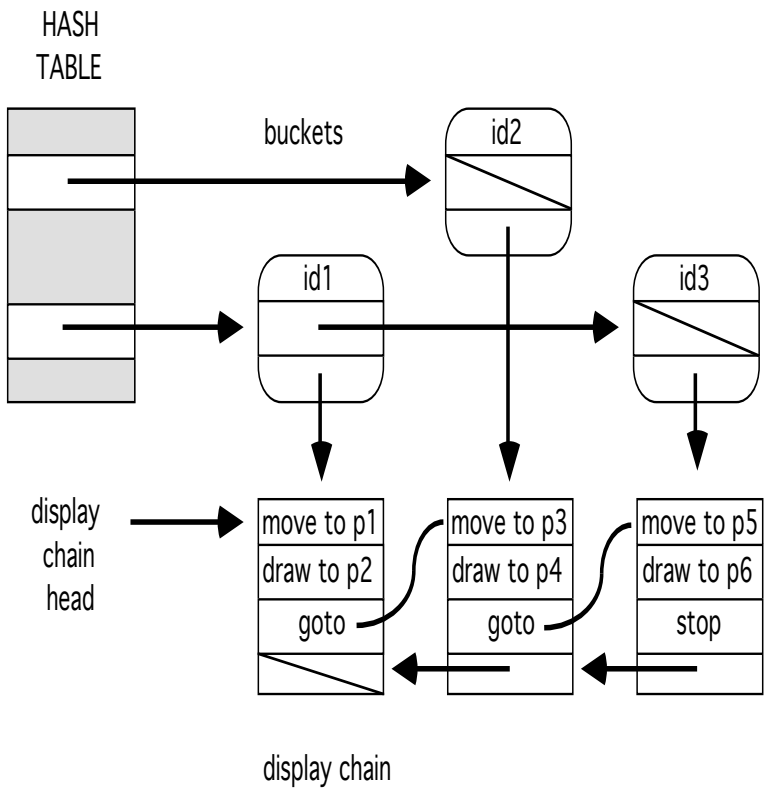
- The design documentation is a programmer's guide to the object
- It defines the coding requirements for the object
- It must include the design of the data structures and algorithms implementing the encapsulated object
  - internal data structures
  - algorithms for each operation
- It must be a faithful representation of the code

Data Structures

- They are specific to a particular realization of the object
- They must stress algorithmic efficiency
- They must be presented at a clean abstract level
- They must be consistent with the external view of the object
- They should be documented in sufficient detail so as to ensure straightforward coding but they need not be coded

Complexity Control

- Performance constraints increase complexity
- Clean presentation
  - simplifies documentation
  - increases dependability
  - facilitates analysis
- Invariant properties are critical to defining data structure integrity



Operations

- No design should be given for trivial operations
- Simple operations should be specified only in terms of pre- and post-conditions
- Complex operations should be specified by providing clearly stated algorithms
- The level of abstraction must be properly matched to the complexity of the algorithm
- An understanding of the data structures design should be assumed on the part of the reader
- The algorithms must be implementable in the target language but should not be tailored to it

Pseudocode Basics

update(picture P)

Find the layer list L associated with the depth of P.

if no such list

then raise exception no_such_picture

endif

Find a picture P’ with the same name as P in list L.

if no such picture

then raise exception no_such_picture

else replace P’ by P

endif

Relations among Objects

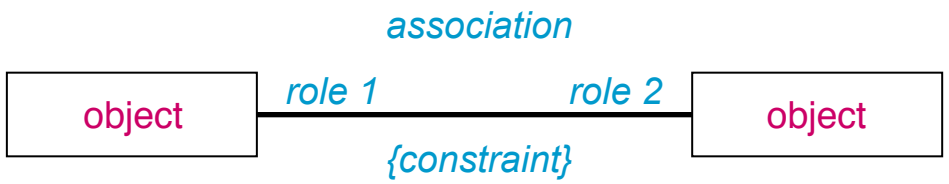
- Objects are not used in isolation
- Complicated designs entail complex relations among objects
- Design documentation, analysis, and verification benefit from a proper understanding of the relations that hold among objects
- Certain relations among objects are induced by the application and relate to semantic consistency
- Others are introduced by the design and relate to various forms of data integrity
- Still others reflect the mechanics of object composition and interaction

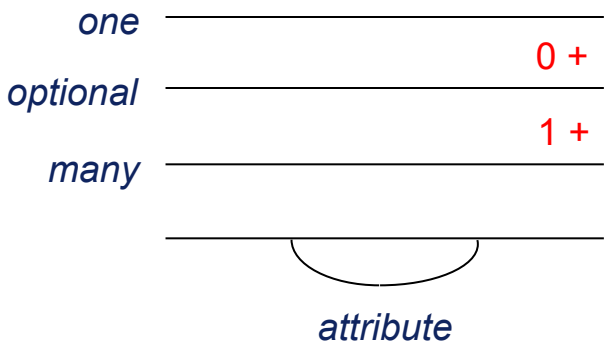
Semantic Relationships

- Semantic relationships capture integrity constraints
  - determined by essential aspects of the application (e.g., each person has a social security number)
  - introduced by the designer (e.g., the number of reserved seats stored in the flight object equals the number of booking records allocated by the reservations object)
- They play an important role in ensuring the correctness of the system
- They can be extended to include external "objects"
- They are captured by the notion of association

Associations

- They capture n-way relations among objects



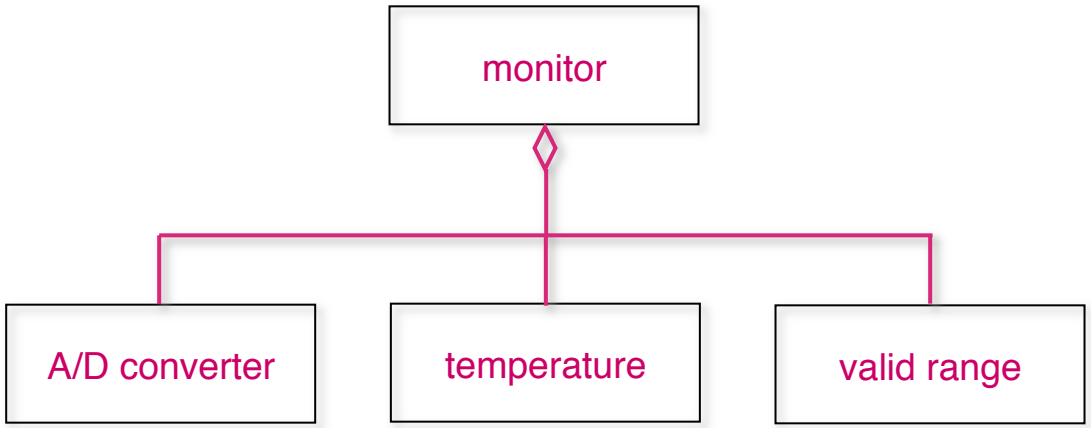


Structural Relationships

- Complex objects can be constructed from simpler ones
- The part-of relation is captured by the notion of aggregation
- It plays an important role in capturing critical aspects of the software architecture, how components are put together
- Good design practices disallow access to the subcomponents by any other objects in the system

Aggregation

- Aggregation is a special relation that
  - captures structural properties of objects
  - enables object composition

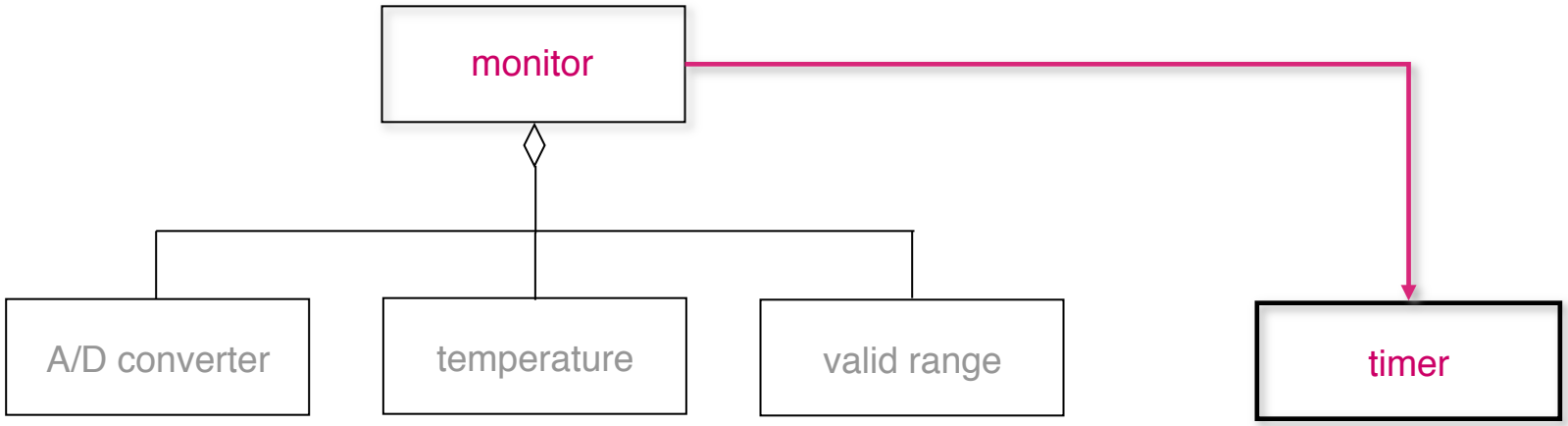


Behavioral Relationships

- Making use of the services offered by an object involves the invocation of operations on that object
- The entity invoking the operation must refer to the object
- The objects one is aware of are called its acquaintances
- Good design practices suggest that an entity ought not to know about anything it does not use
- The reference relation assumes that any object that is known is potentially accessed
- It is usually the case that a composite object references its immediately subordinate objects—there is some redundancy with the aggregation relation in this case

Reference

- Reference is another specialized relation that
  - captures run time usage of objects
  - constrains the method invocation pattern

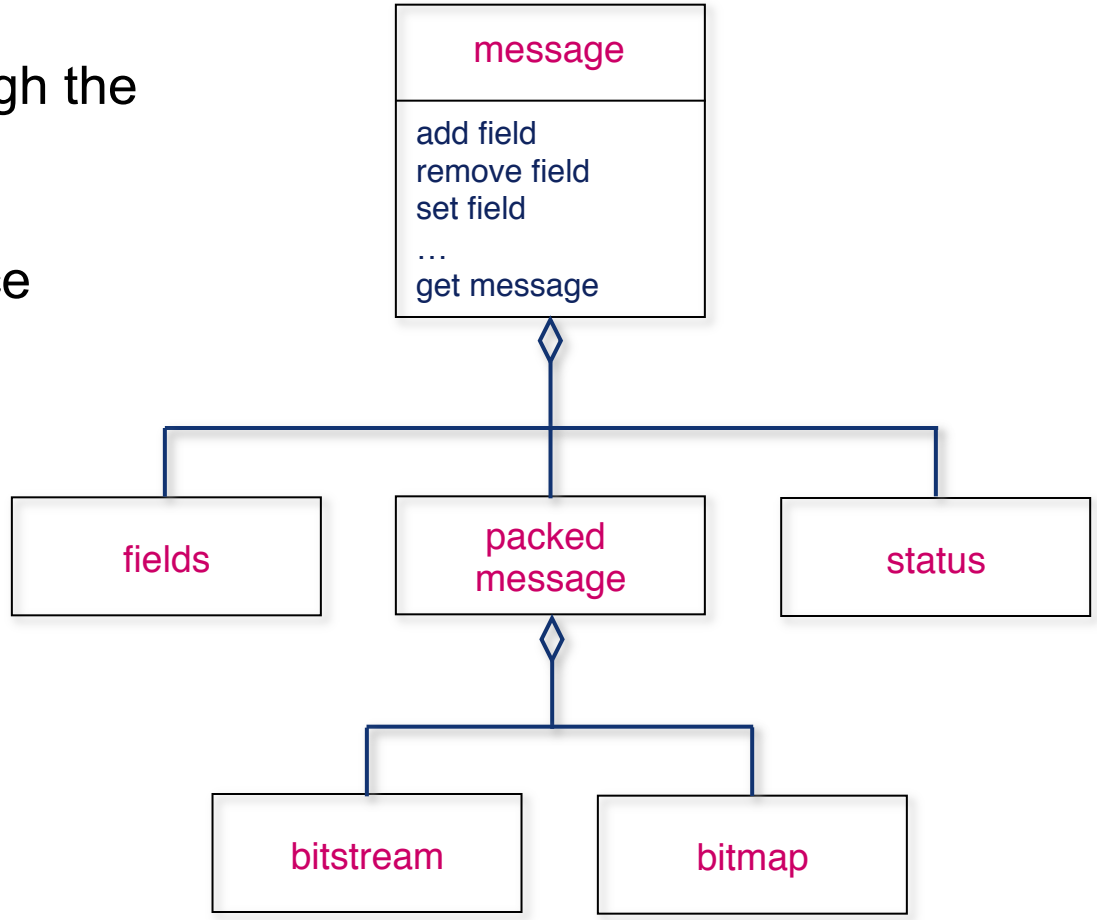


Object Composition Patterns

- Objects must be designed such that they
  - are simple to understand
  - provide a high degree of encapsulation
  - offer maximum independence among objects
- One must exercise design discipline in order to preserve these advantages while
  - faced with the complexities of real systems
  - interfacing with code which is not object-oriented
  - having access to complex and unstructured object composition methods

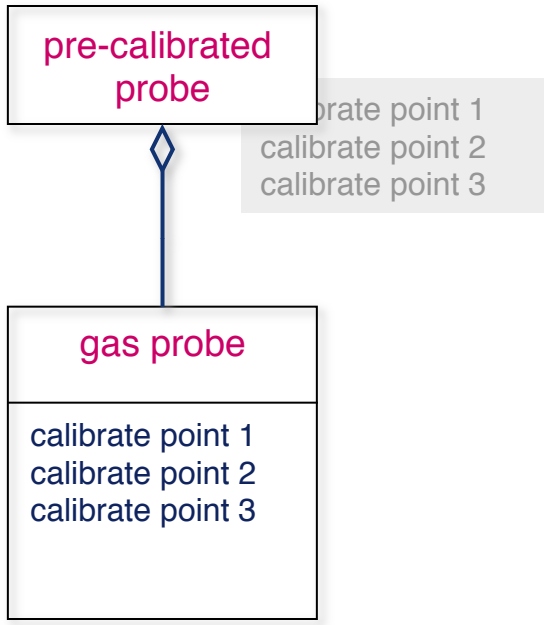
Nested Objects

- Nested objects are constructed strictly through the use of aggregation (tree structure)
- Each object can reference only its subcomponents
- It is desirable for sibling objects to be of similar complexity and level of abstraction



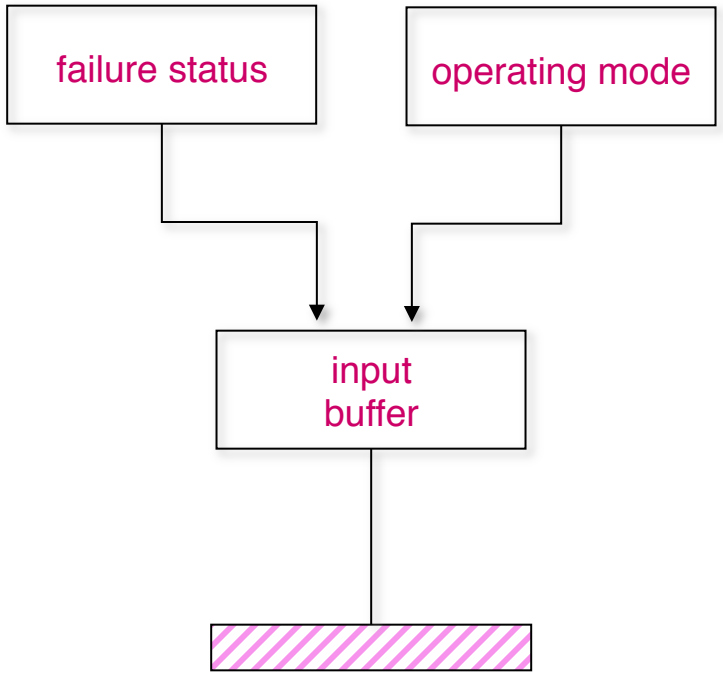
Hiding

- Hiding is an abstraction mechanism
- Hiding allows one object to project out some of the operations provided by its components
- The operations that are not hidden are delegated to the subcomponents
- A general purpose capability is tailored to a specific application



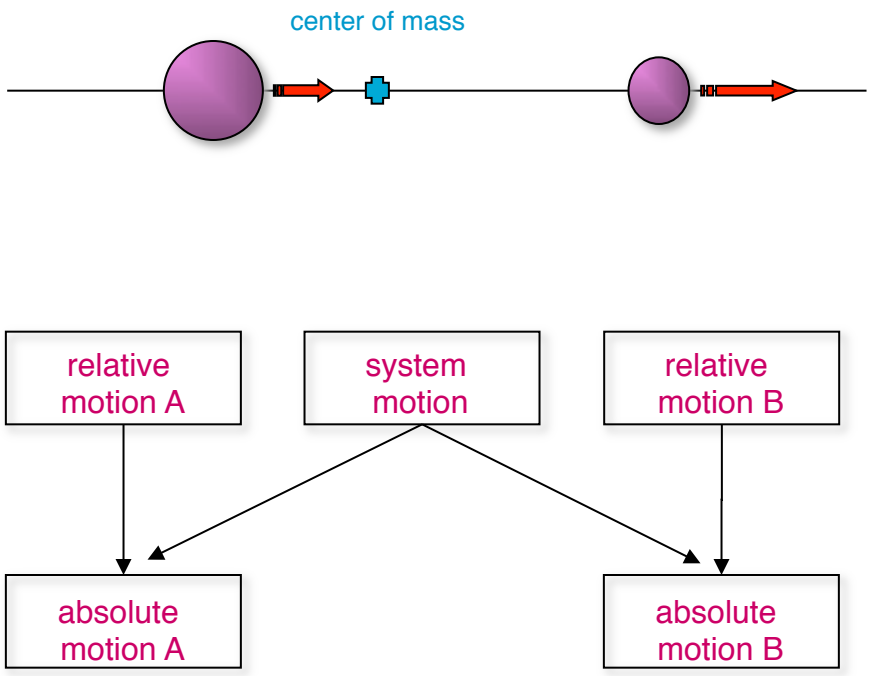
Transparent Sharing

- Object sharing occurs when two or more objects have an acquaintance (reference) in common
- Transparent sharing occurs when none of the objects involved can detect that sharing takes place
- This is often the case when one physical interface supports several logical interfaces



Mutation

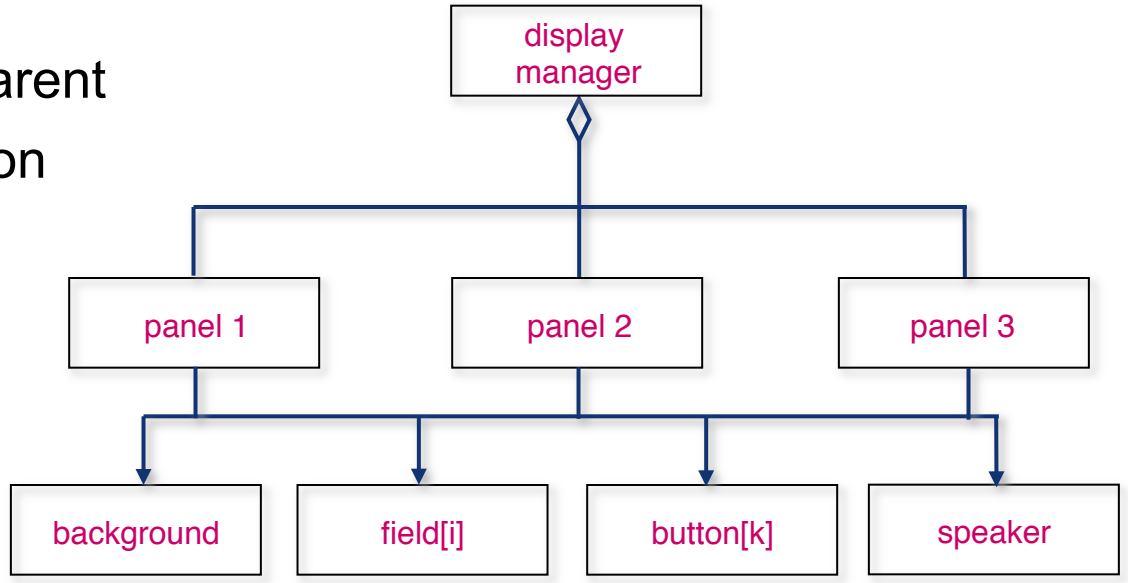
- Mutation is an abstraction pattern that relates two object layers
- It involves a change in the encapsulation of the composite state of the lower level in response to the needs of the upper levels in the design
- It is especially helpful for restructuring low level physical interfaces into more abstract ones
- The sharing of the lower level objects must be transparent



Layered Objects

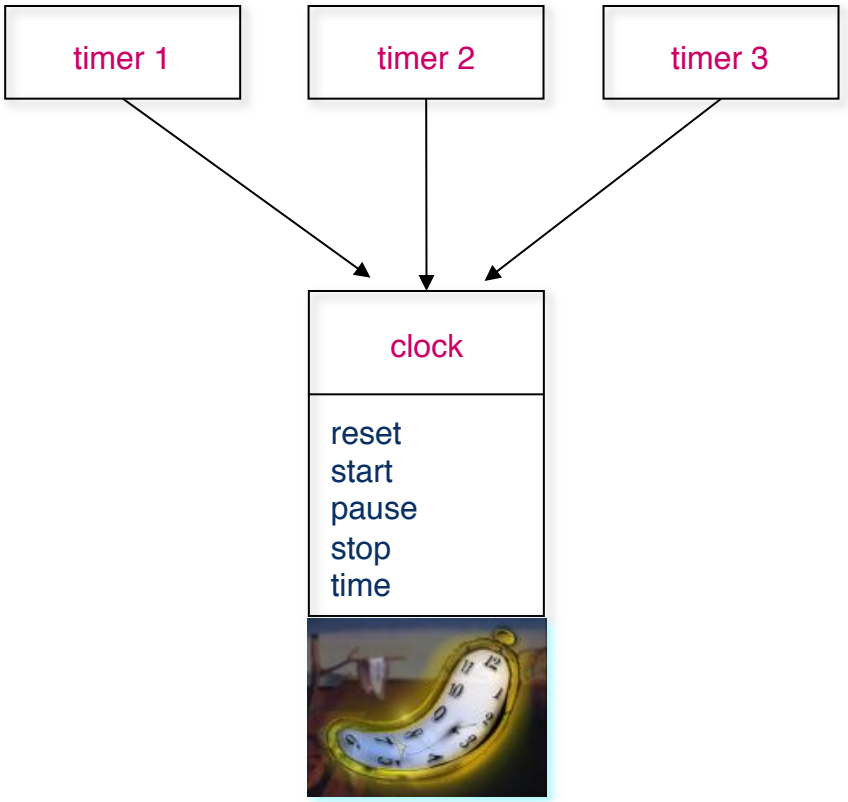
- A layered object consists of a hierarchically organized set of objects
- An object at one level can reference all objects on the level below
- Sharing is not transparent
- The level of abstraction decreases with depth

- *Dynamic restructuring*
- *One panel is active at a time*



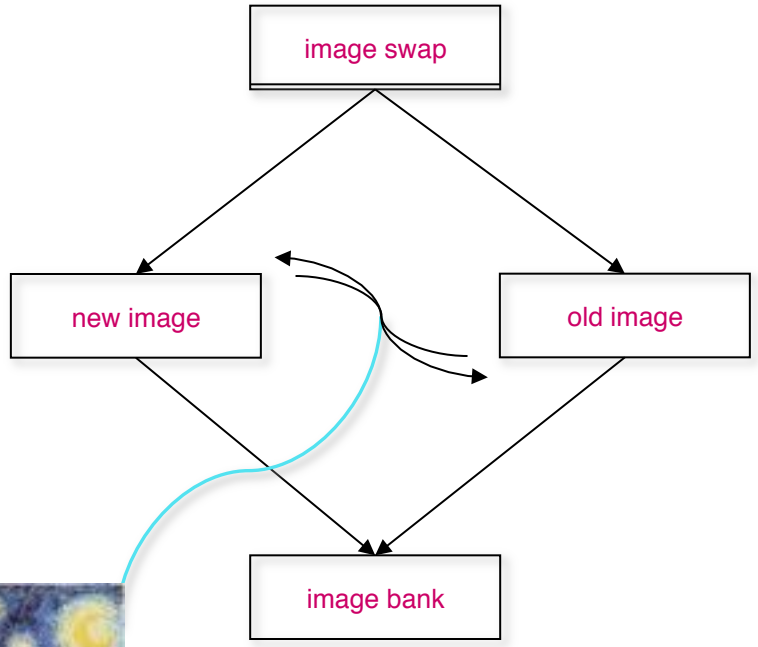
Shared Resources

- Object sharing is highly undesirable
- When sharing cannot be avoided it should be minimized, structured, and made uniform



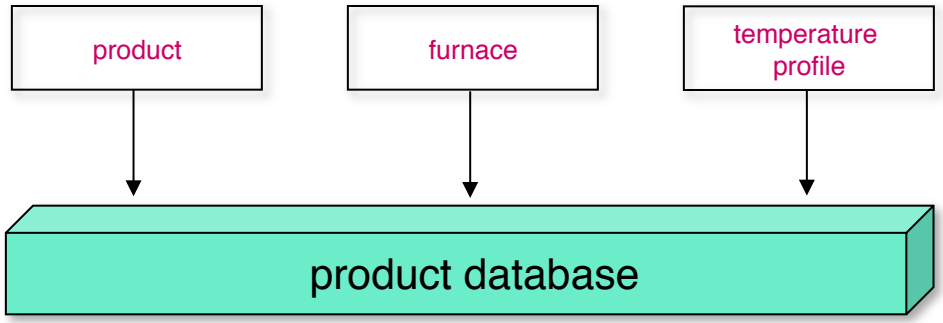
Shared Implementation

- Performance considerations often require objects which are essentially independent to be encapsulated in a single object managing their implementation
- The desire for generality may also lead to shared implementations



Object Veneer

- Legacy code need not be an impediment in the application of object-oriented design
- Existing code can be encapsulated as a set of objects which are available to the remainder of the system

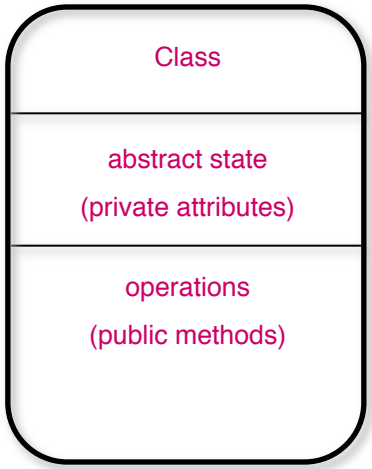


Object Generalization

- The class is a programming language construct used to extend an existing language with user-defined types
  - facilitates type checking
  - promotes encapsulation and modularity
  - encourages code reuse
  - reduces code development
- The class is an abstract concept used to characterize the set of all objects having a common signature
  - facilitates analysis
  - promotes a high-level design
  - encourages design reuse (design patterns)

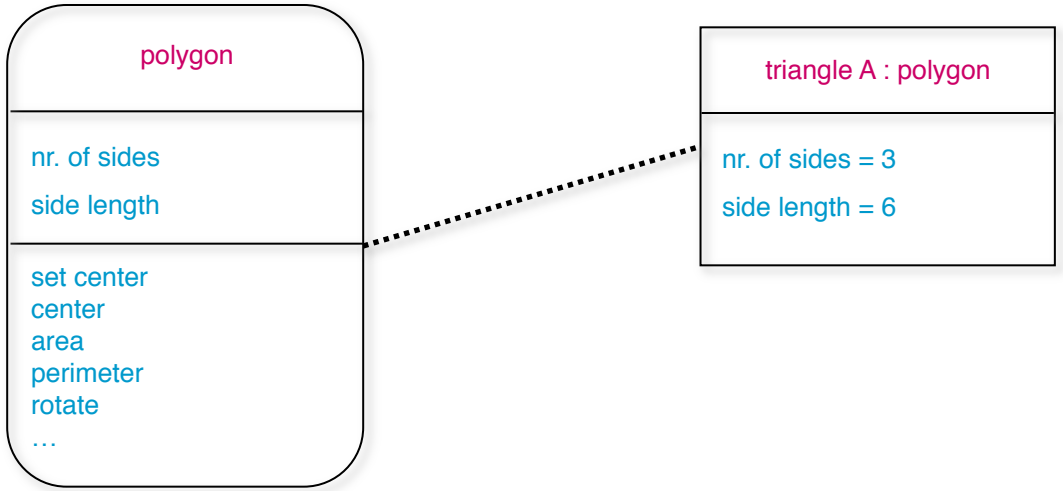
Class Definition

- A class is characterized by
  - a unique name
  - private attributes and their types (internal abstract state)
  - public methods or operations and their respective signatures (name, argument and return value types)
  - semantics of each method
- In the absence of a class concept, a new (identical) definition would be required for each object of the same type



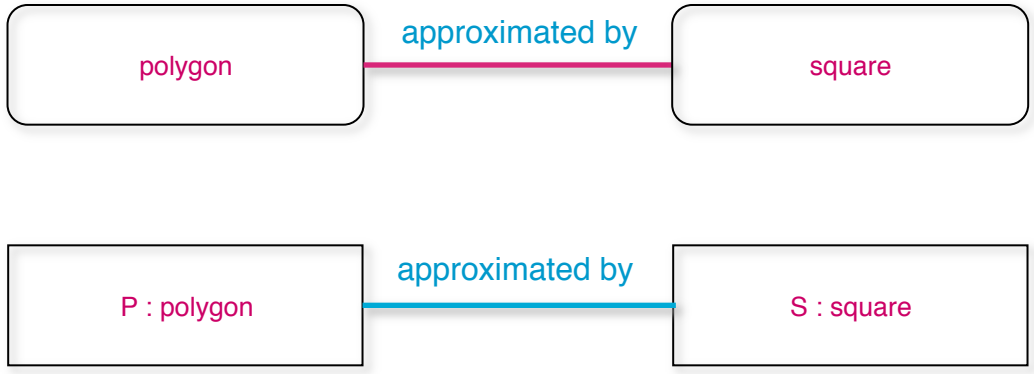
Instantiation

- The relation between a class and an object of that class is called instantiation
- Many similar objects used in the design are documented only once



Relation Transference

- Some of the relations among classes can be instantiated as relations among objects

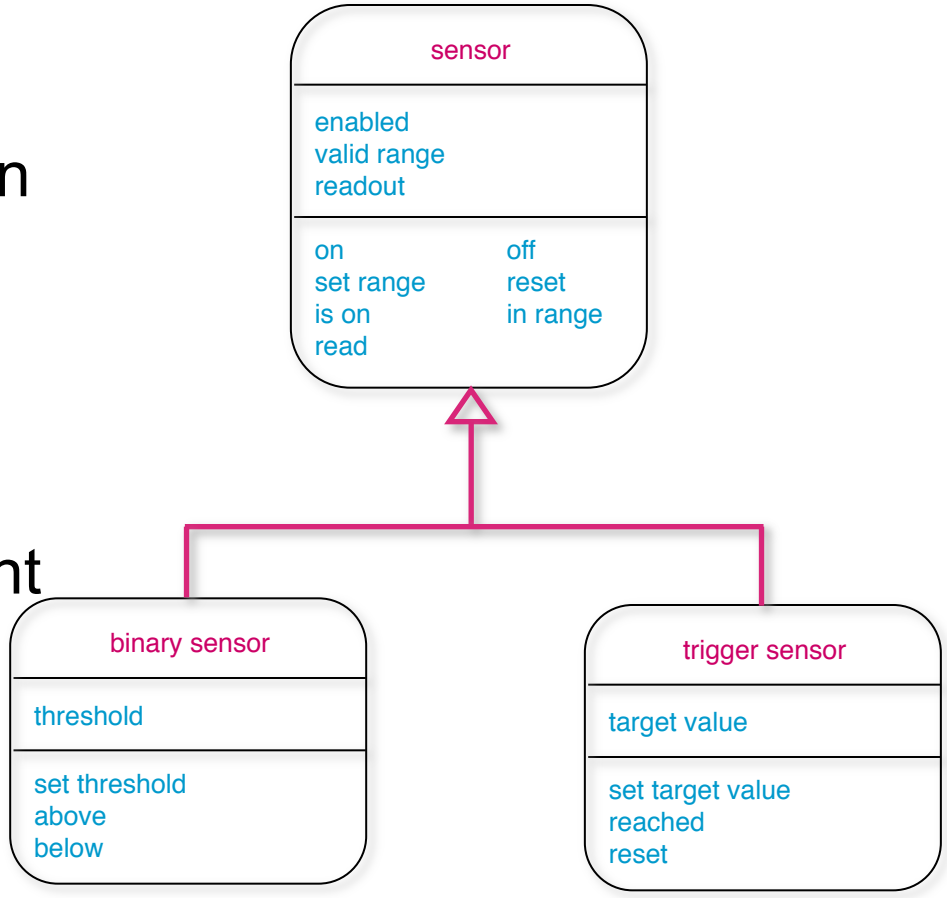


Inheritance

- New classes (subclasses or derived classes) can be defined from existing ones (superclasses or base classes)
- The subclass definition consists mainly of
  - the name of the superclass
  - the name of the class
  - additional attributes
  - additional operations
- Additional redefinition capabilities are
  - semantic redefinition of selected operations (constrained!)

Design Implications

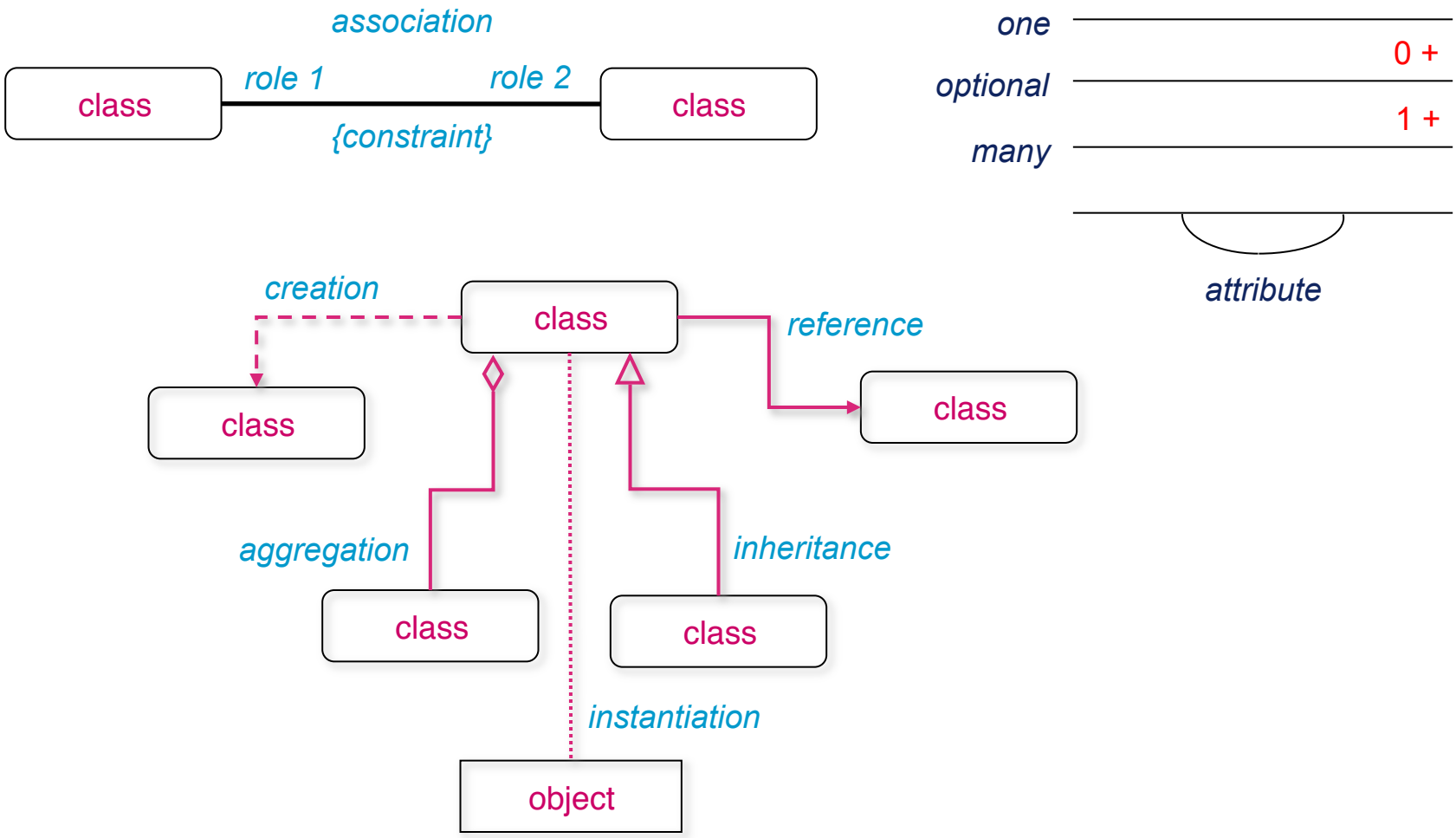
- Reduction in documentation size
- Simpler conceptualization of object definitions and their interrelations
- Effective path towards generalization
- Reduction in development time



Managing Object Complexity

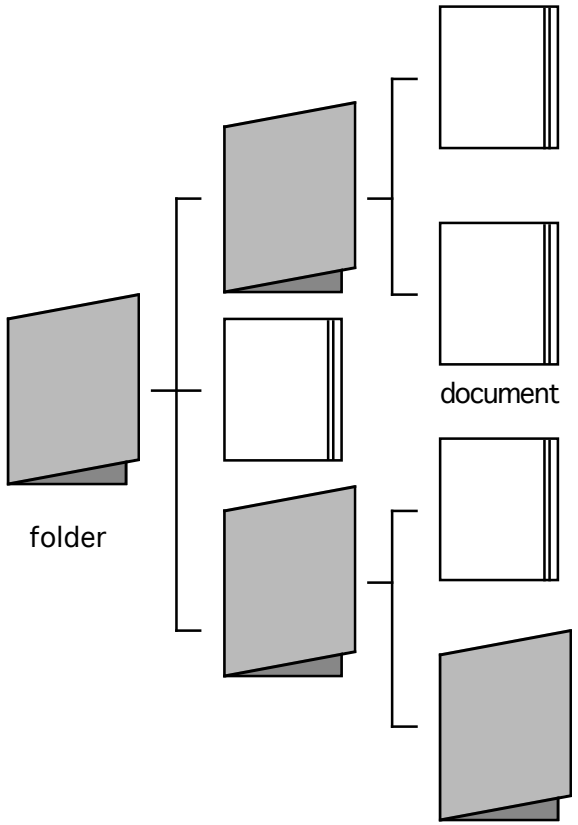
- Complicated objects require powerful object definition capabilities
- Complexity is moved to the level of class definitions
- Class diagrams employ relations such as inheritance, aggregation, reference, and more to facilitate the definition of complex classes
- Many fundamental relations among classes can be captured in diagrammatic form
- Class diagrams play an important role in object-oriented analysis
  - they do not define the architecture
  - they are ancillary meta level specifications of components (and connectors)

Class Diagrams



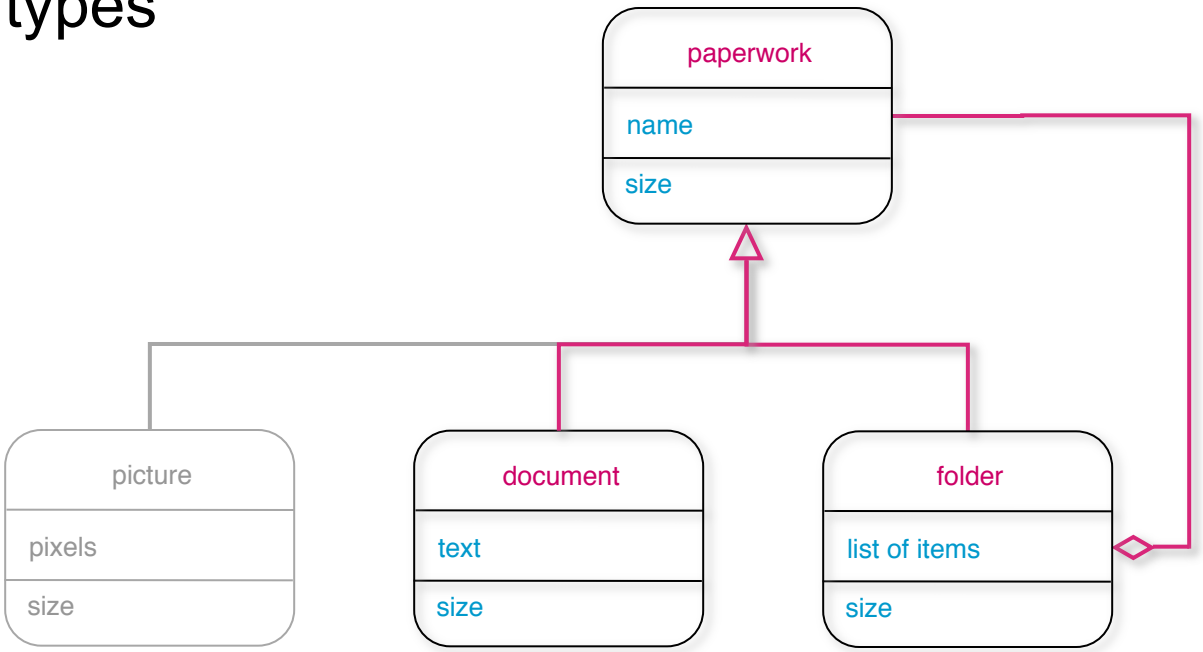
Recursive Object Structures

- Consider an application in which a folder must be able to hold other folders as well as simple documents
- The key operations needed are
  - find by name both folders and documents
  - count the total number of items in the folder



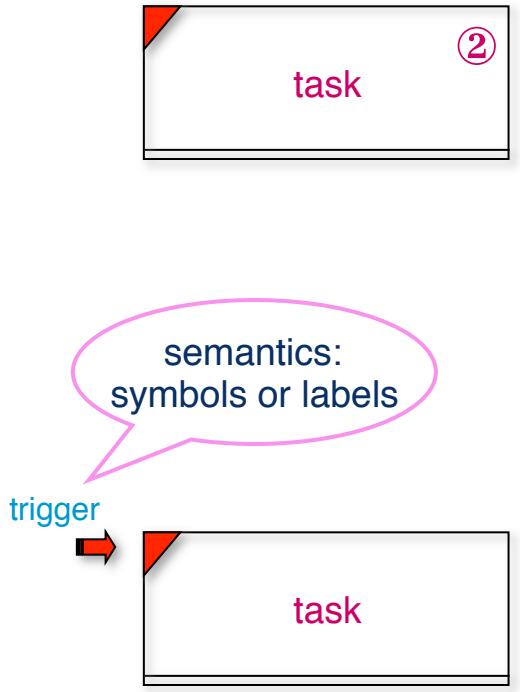
Sample Solution

- The generalization demanded by this problem accommodates future document types



Task

- A task is a sequential process having its own independent thread of control
- Its execution results in the invocation of methods on objects in the system
- Task execution may be
  - periodic, scheduled at a given priority
    - lower number represents a higher priority
  - reactive, in response to some actuator



Task Activation

- Each actuator is associated with some trigger condition
- Trigger conditions must be defined outside the diagram
- Different notations should be used for different triggers
- Implementability of the trigger mechanism must be established
- Actuators can hide complex implementation details and simplify design
- Sample trigger conditions
  - clock signal
  - timer overflow
  - system event (e.g., failure, power up)
  - application event
  - message arrival
  - system condition that can be independently assessed

Concurrency Implications

- Multitasking is both necessary and convenient
- Complicates design analysis
  - starvation due to deadlocks or priority inversion
  - missed deadlines or low performance
- Requires an understanding of
  - fundamentals of concurrency
  - system characteristics
- Special notation needs to be introduced to help design analysis

Semantics of Concurrency

- Certain concepts are fundamental to any form of concurrent processing
- In the absence of a clear definition for these concepts both design and programming become impossible

x := y +1

; or || or []

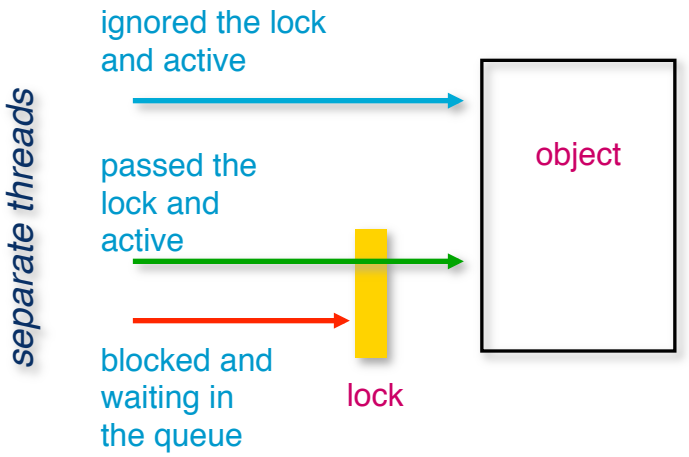
y := x + 1

- Atomicity addresses the issue of minimal guarantees of non-interference among operations
- Fairness is concerned with minimal assumptions that can be made about the scheduling of operations
- Synchronization deals with the mechanics by which concurrent operations coordinate their executions

Synchronization

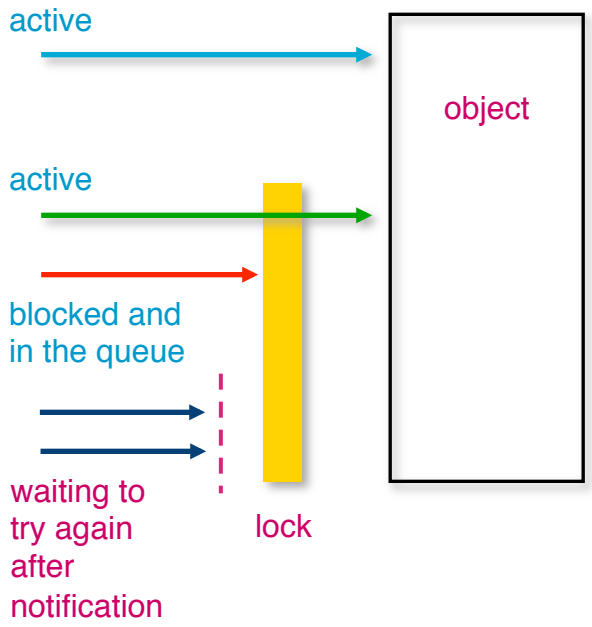
- Synchronization defines a structured mechanism for coordination among tasks
- Mutual exclusion is a general synchronization mechanism implementable in most systems
- The actual mechanics of synchronization are language specific (e.g., Ada tasks, Java synchronized methods)
- At design level it is convenient to express synchronization by specifying mutual exclusion requirements

- Java model



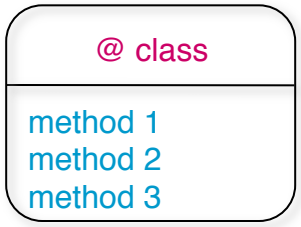
Java Notification

- The `wait(object)` operation places the thread on the wait queue
  - the object lock is released
- A time period may be specified and the thread is removed from the wait queue when the time interval expires
- The `notifyAll(object)` operation removes all the threads from the wait queue
- In both cases the released threads must be scheduled and must acquire the lock

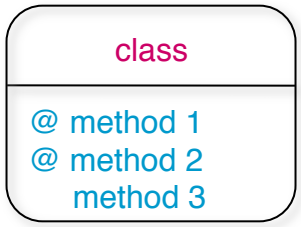


Object Level Synchronization

- The notation is the same for classes and objects and so are the semantics
- Synchronized objects specify mutual exclusion among all methods

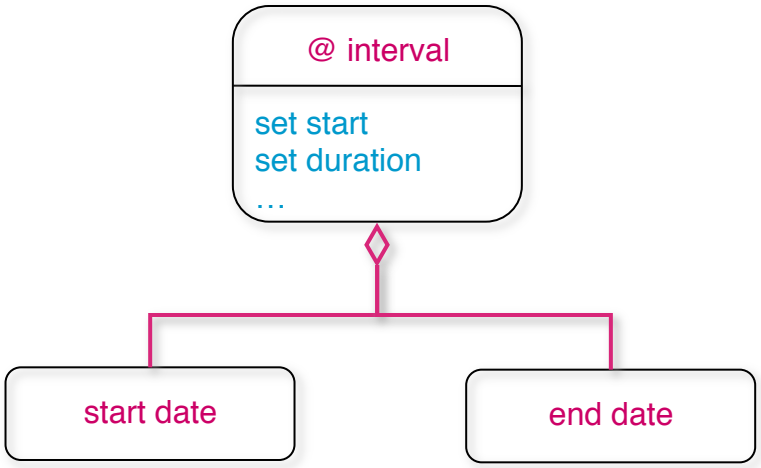


- Notation and semantics may be adjusted for different settings
- Synchronized methods limit mutual exclusion to identified methods

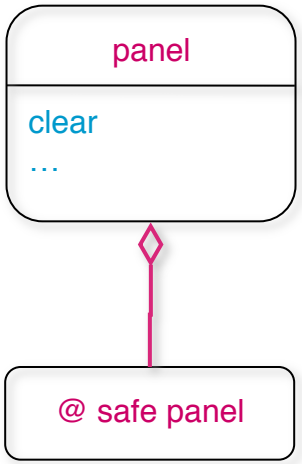


Expressiveness

- Containment

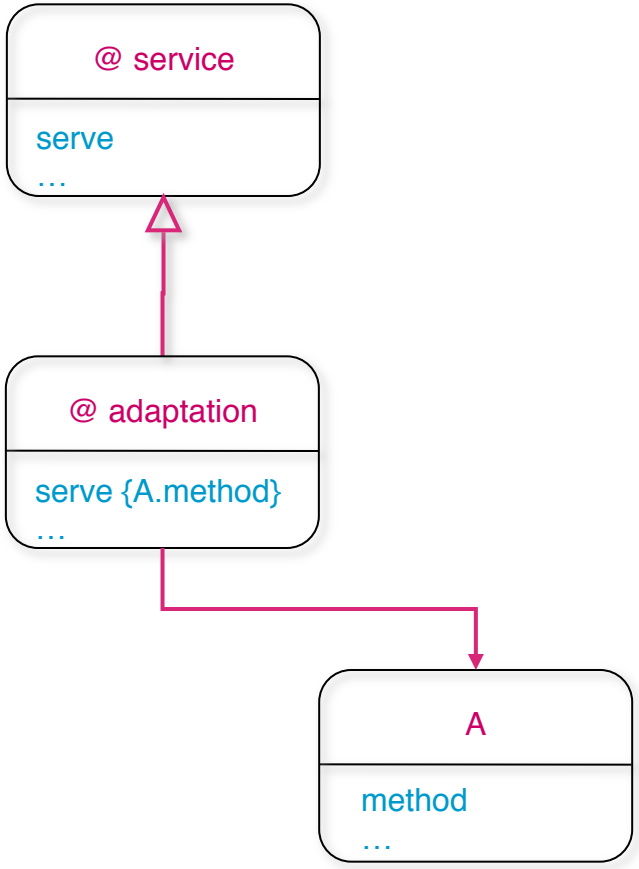


- Hidden synchronization

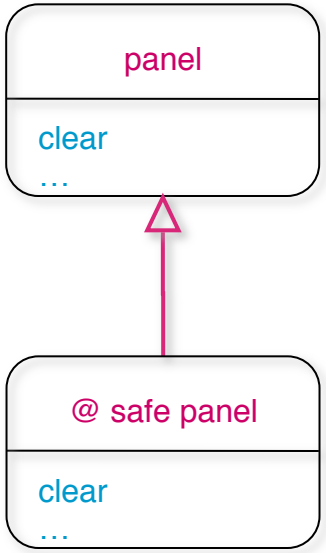


Expressiveness

- Adapter

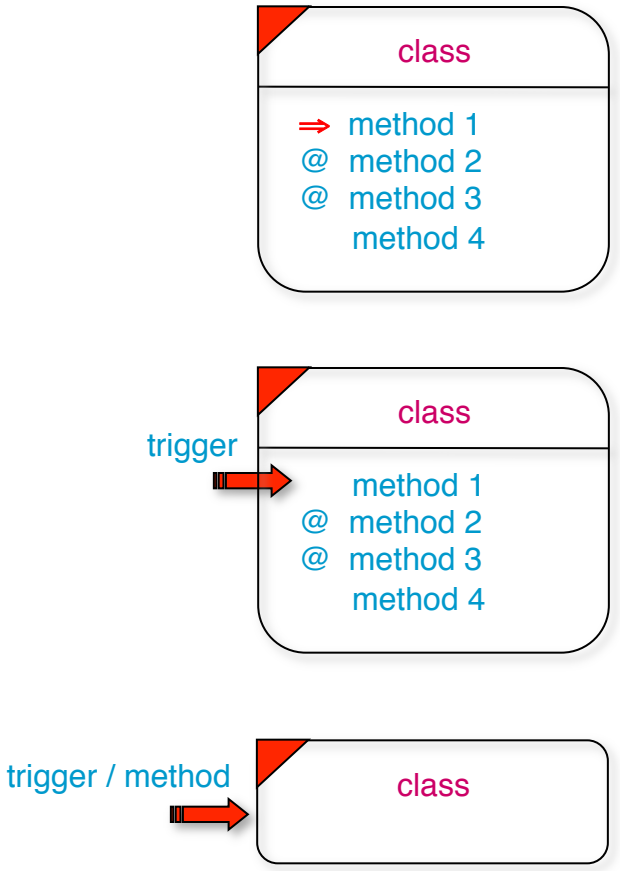


- Synchronized wrapper



Active Objects

- Exhibit behaviors equivalent to wrapping a task inside an object veneer to control the execution of a specific method
- The execution of a thread inside an active object may be periodic or reactive



2.5.2 Connectors

- Connectors relate components appearing in the design diagram
  - allow components to interact
  - constrain the nature of their interactions
  - simplify the design by abstracting complex patterns of interaction
- Connectors are specialized components
  - each component is an instance of a class
  - standard class definition methods are used
  - specialized symbols are employed to simplify design

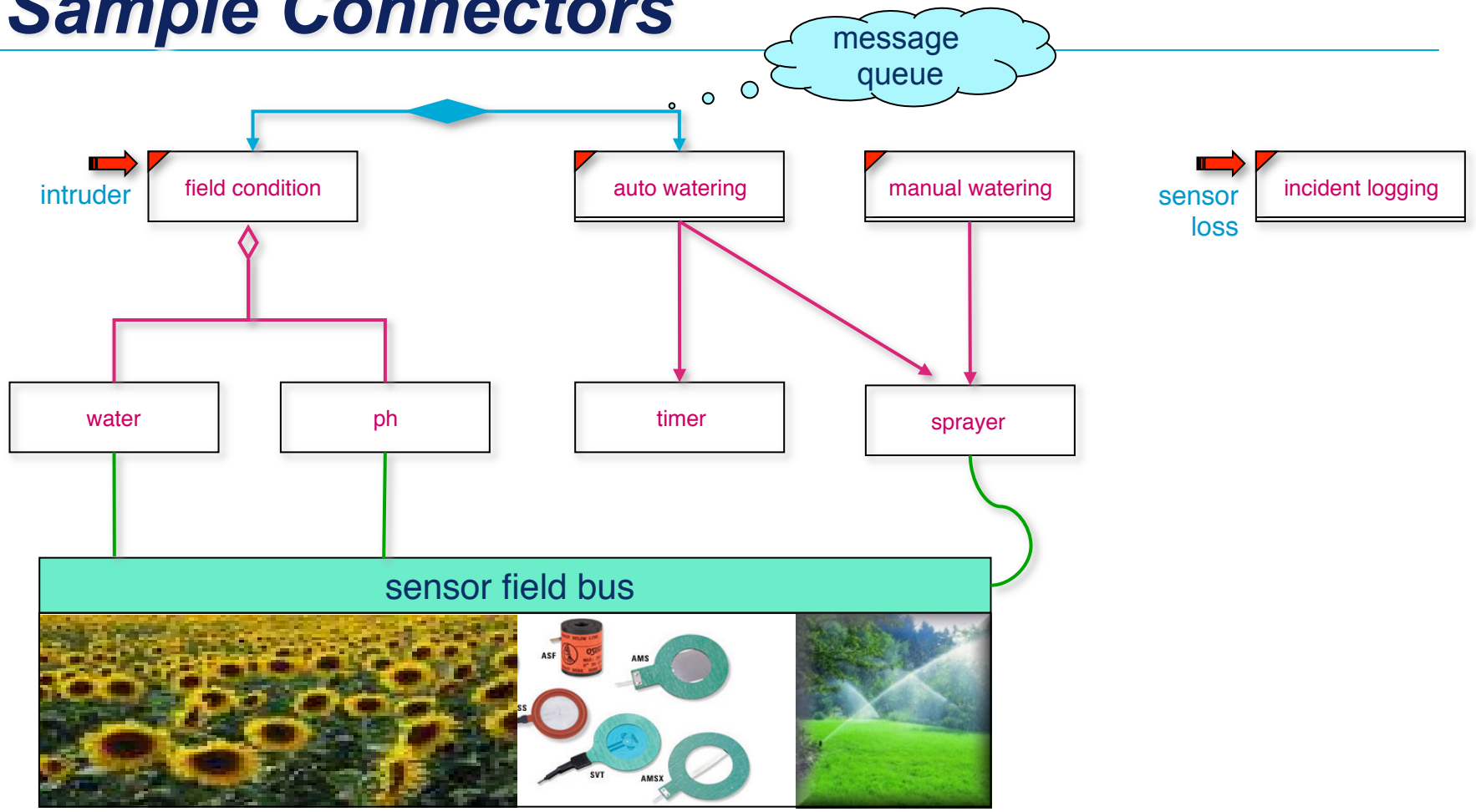
Connector Types

- Basic connectors
  - reference relation constrains method invocation patterns
  - aggregation relation is also a restricted form of reference
- Complex connectors
  - can be two-party or multi-party
  - may be specialized for a particular design
  - may use custom graphical representations
- Transparent connectors have no symbol tying the points of origin and processing
  - event notification
  - publish/subscribe

Connector Instances

- Message passing interactions
  - channels (1-1)
  - ports (n-1)
  - mailboxes (n-m)
- Software bus
- Remote procedure call
- Shared tuple spaces
- Events

Sample Connectors



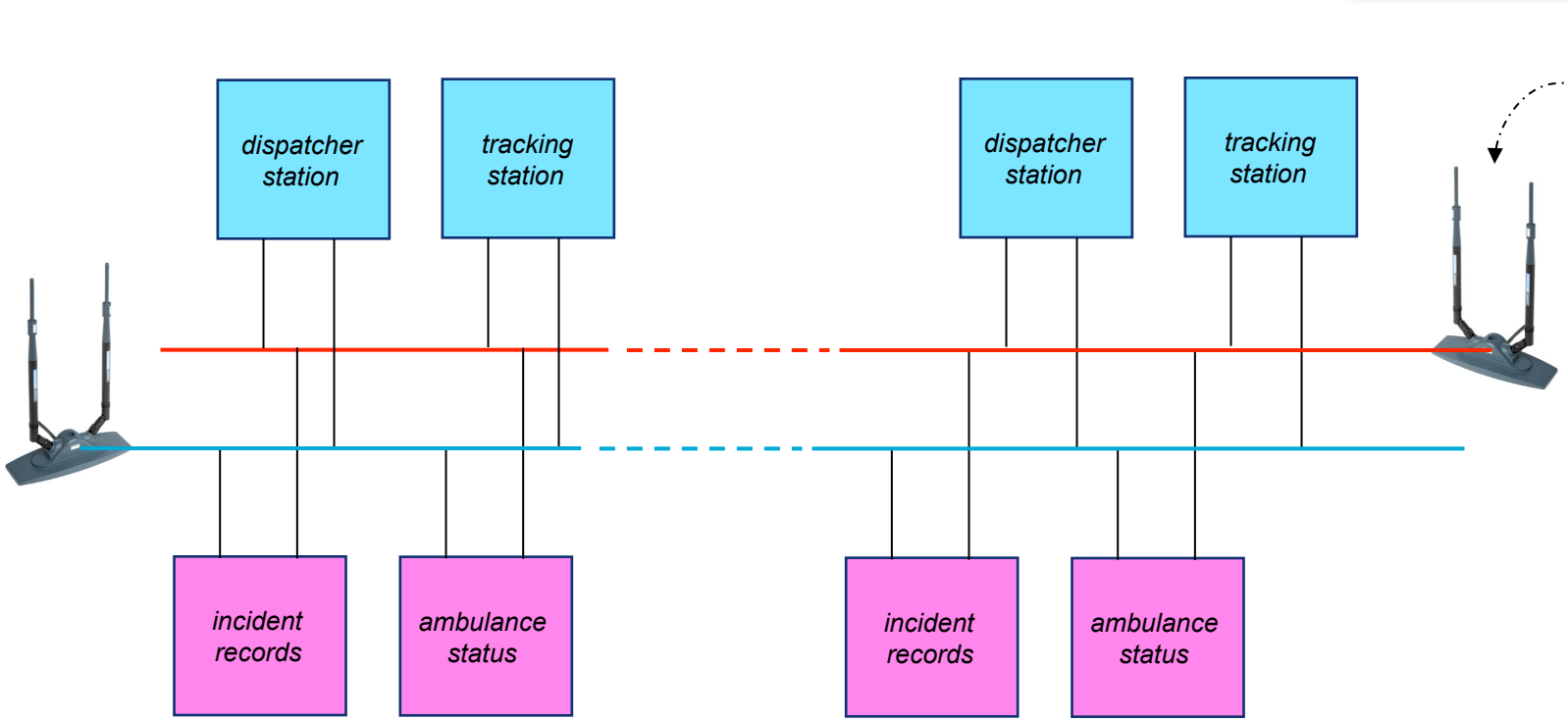
2.5.3 Design Diagrams

- The design diagram is an abstract view of the system structure and key behavior constraints
- The design diagram identifies the key software system components and the relevant connectors
- Components and connectors are defined outside the scope of the diagram
- Annotations allow one to communicate additional information for which there is no precise notation
  - implementation directives
  - timing constraints
  - software binding requirements (COTS)

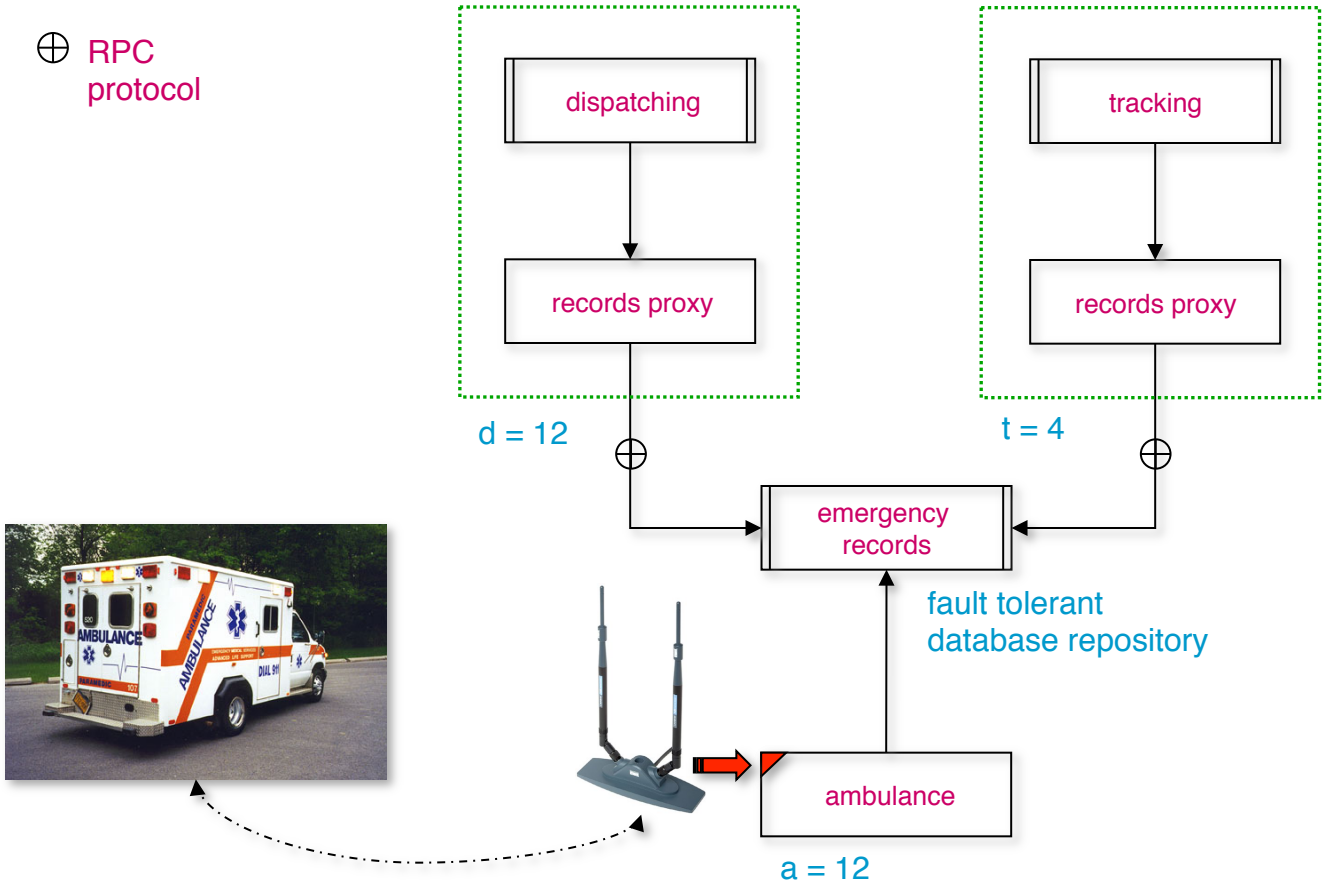
Cost Effective Design Process

- Effective design
  - focuses on early investments in creative thinking
  - delays documentation until stability is reached
- Typical stages in architectural design
  - initial diagram layout
  - diagram refinement and informal object specifications
  - informal design review
  - specification of object interfaces
  - analysis of critical design issues
  - preliminary design review
  - complete specification
  - formal architecture design review

Architectural Sketch



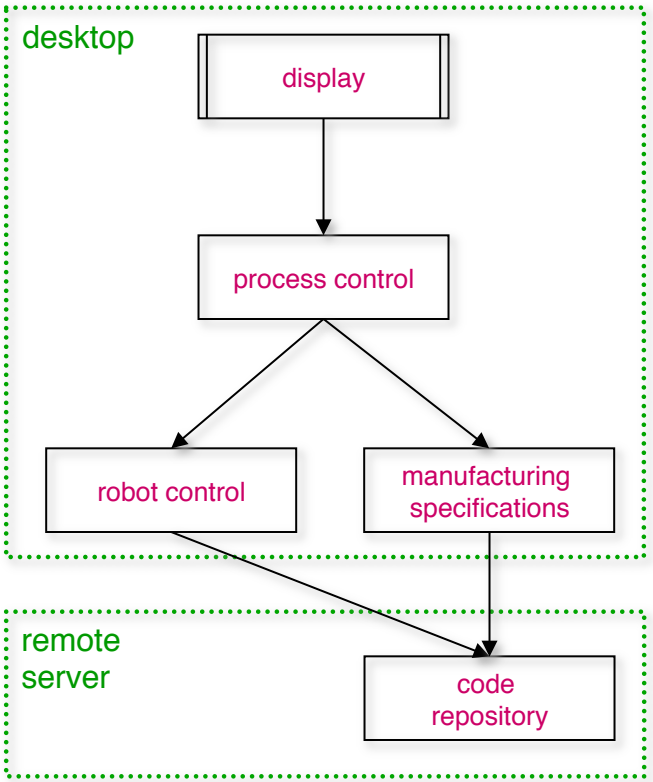
Architectural Design



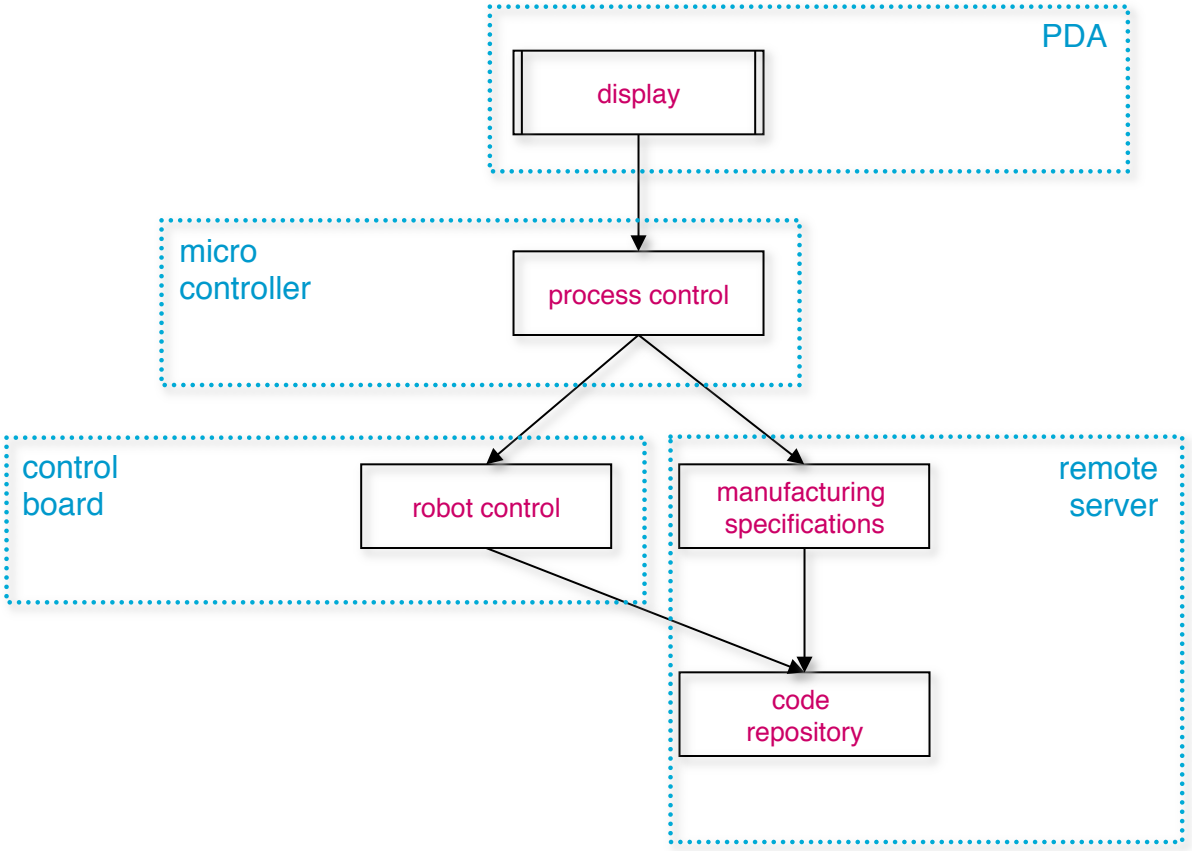
Software/Hardware Allocation

- The allocation of software components to the underlying hardware is
  - a defining feature of the architecture
  - a critical design decision affecting
    - quality of service
    - analysis and evaluation
    - software interfaces
    - coordination/communication protocols
    - maintenance procedures
- If the allocation is explicit (static or dynamic)
  - it can be captured in the form of annotations
  - each software components is mapped to exactly one hardware component
- If the allocation is transparent
  - it requires no explicit graphical notation
  - it affects analysis and evaluation

Allocation Alternatives



Allocation Alternatives

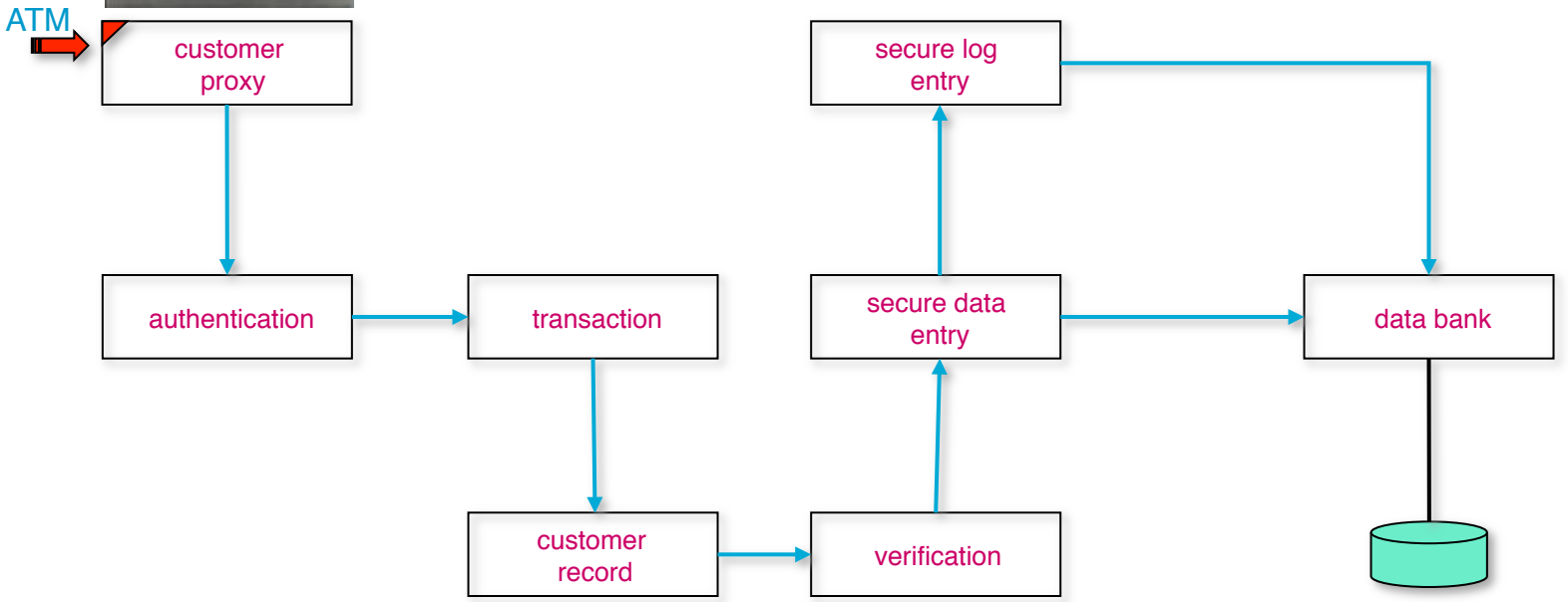


Complexity Control

- Packages may be used to define subsystems (rather than code management)
- Simple interfaces among subsystems facilitate hierarchical decomposition of the system
- Decoupling among components facilitates partitioning of the system
- Multiple views, while difficult to maintain consistent, may be critical in support of analysis
- Typical configurations can assist in providing a finite and comprehensive description of a dynamic system

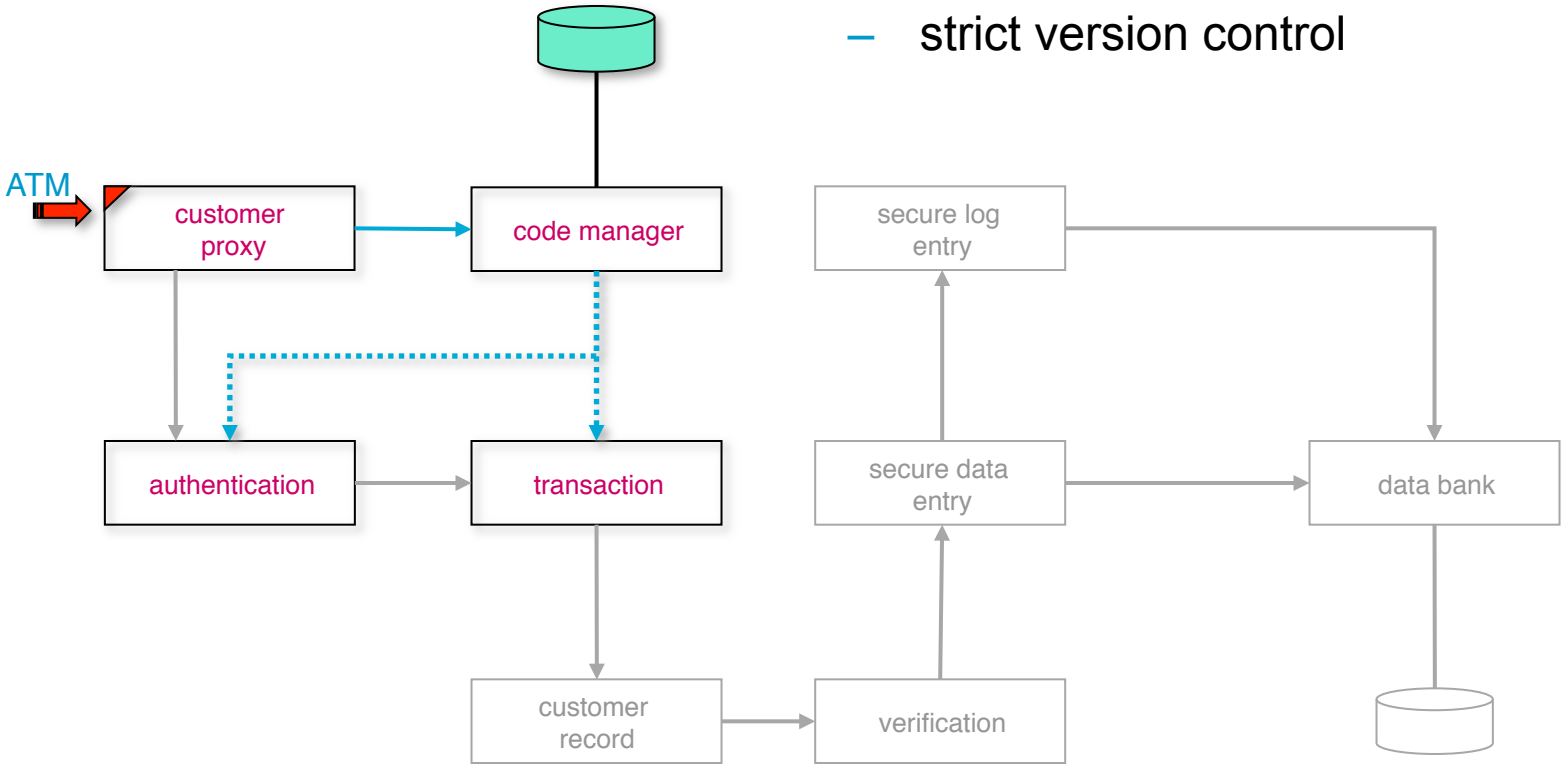
Multiple Views

- Customer service
  - sample partition



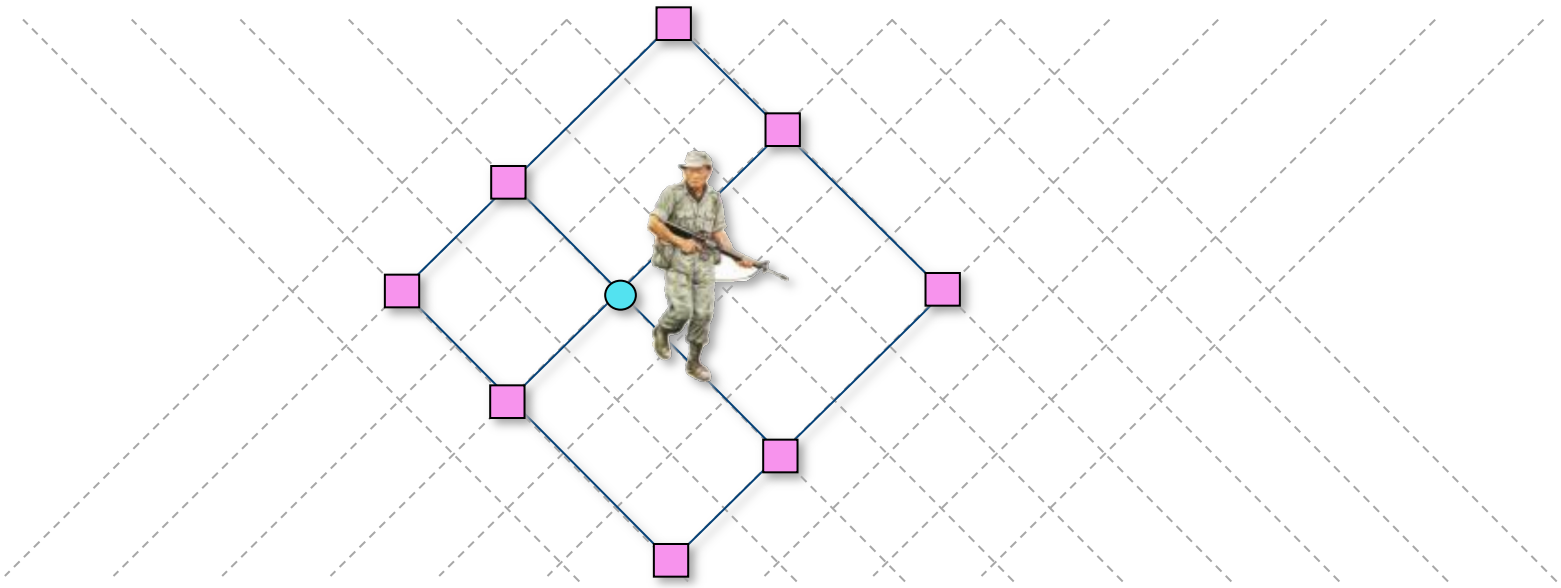
Multiple Views

- Code management
  - alternate viewpoint
  - strict version control



Dynamic Structures

- Sensor grid protection
  - mobile agent based application
  - agent cloning maintains the structure



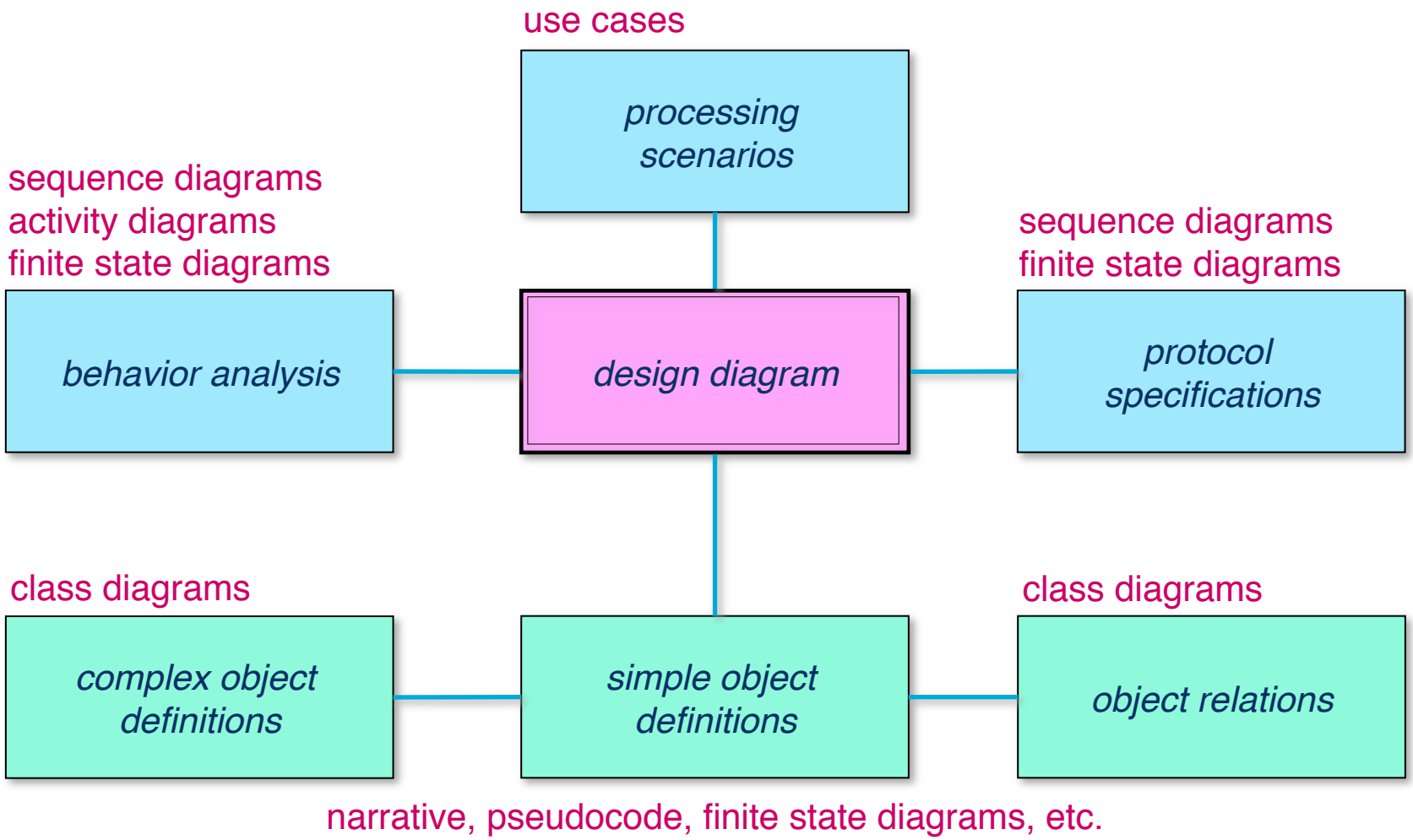
2.5.4 Meta Level Specifications

- A strict definition of the design diagram
  - enables rapid and cost effective design
  - focuses the design process on fundamentals
- Component specifications complete the picture by providing
  - precise interface specifications
  - exact behavior specifications
- Meta level specifications provide additional support for
  - definition
  - analysis
  - specification

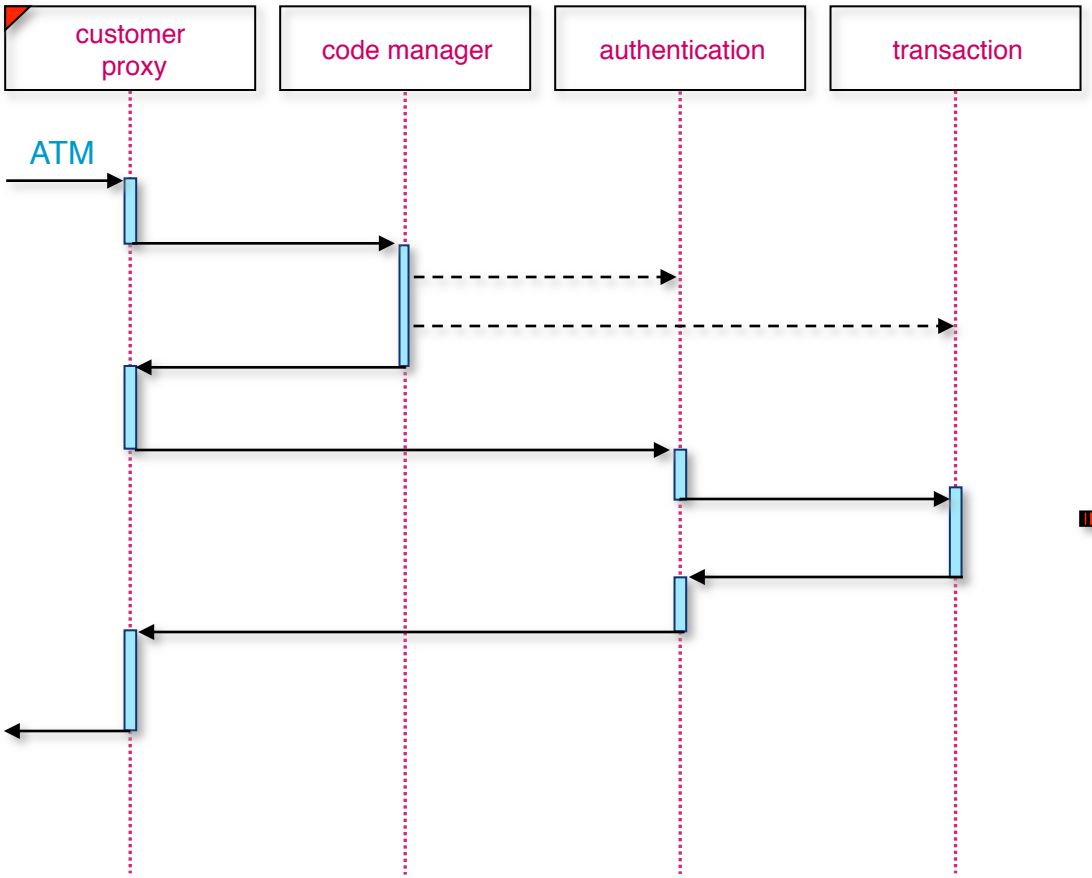
Impact on Architecture Design

- Simplify the object definition process
  - class diagrams
- Capture fundamental relationships among objects
  - class diagrams
- Define interaction protocols
  - sequence diagrams
- Formalize behavior patterns for analysis or specification
  - sequence diagrams
  - activity diagrams
- Capture interaction or processing scenarios
  - use cases
  - activity diagrams

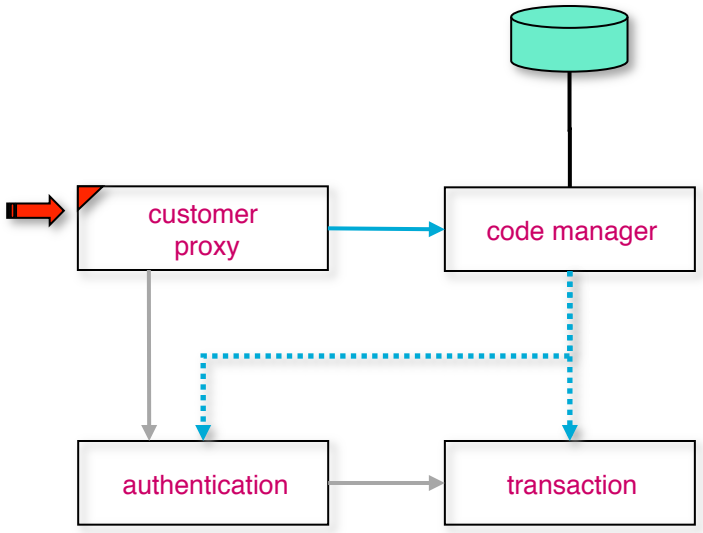
Documentation Revisited



Sequence Diagram

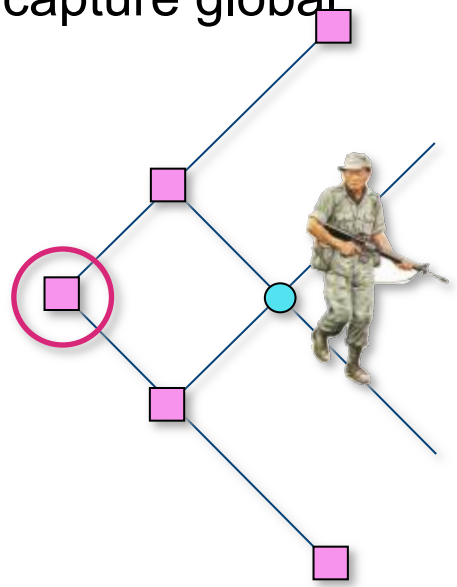


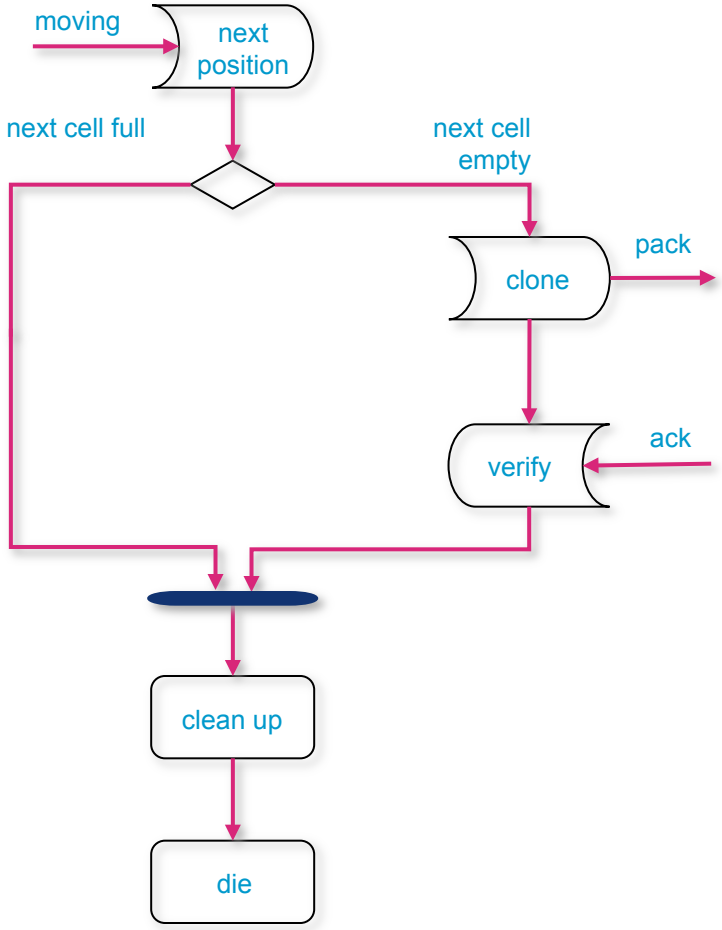
- Is the most recent code being used?
- Can authentication be bypassed?
- etc.



Activity Diagram

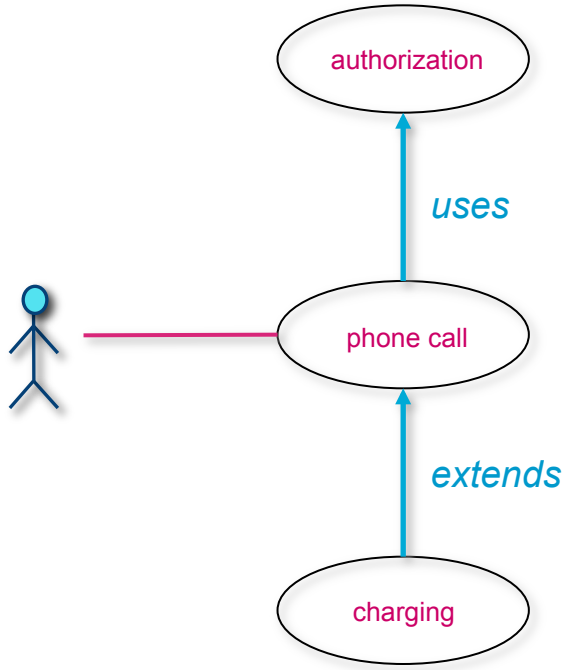
- An activity diagram is a specialized finite state machine—action termination controls transitions
- When used to describe the behavior of one object, all actions are local
- When used to capture global behavior, actions are distributed





Use Cases

- Actors
  - model the environment and the users
  - initiate activity by providing stimuli
  - can be primary or secondary
- Use cases
  - are complete courses of action initiated by actors (basic or alternative)
  - can be extended by (interrupt analogy) or use other use cases (call analogy)



Conclusions

- The development of a software architecture is a complex task
  - multiple perspectives
  - different notations
  - significant and sophisticated analysis
- The complexity of the notation and concepts should be minimized
- The design process should focus on fundamentals and on the creative activities foremost
- Precise and complete documentation is paramount but not the dominant activity