Design - interfaces and components

What is an interface?

design by contract

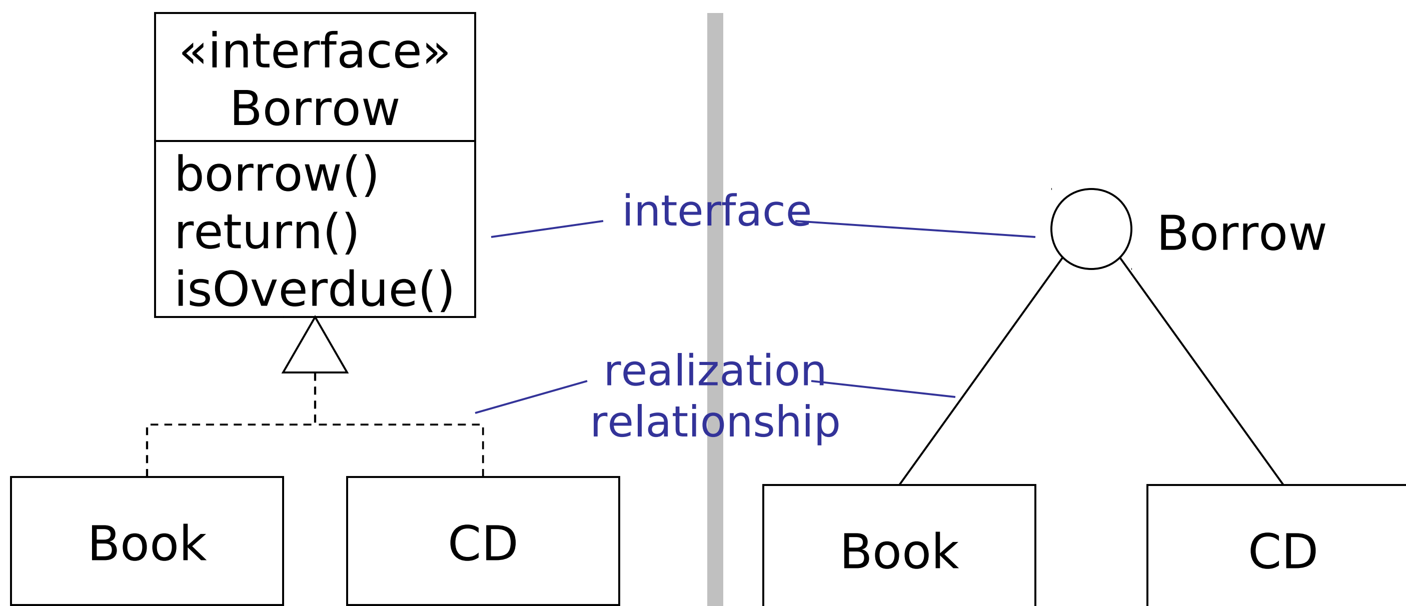
- An interface specifies a named set of public features
- It separates the specification of functionality from its implementation
- An interface defines a contract that all realizing classifiers *must* conform to:

Interface specifies	Realizing classifier
operation	Must have an operation with the same signature and semantics
attribute	Must have public operations to set and get the value of the attribute. The realizing classifier is not required to actually have the attribute specified by the interface, but it must behave as though it has
association	Must have an association to the target classifier. If an interface specifies an association to another interface, then the implementing classifiers of these interfaces must have an association between them
constraint	Must support the constraint
stereotype	Has the stereotype
tagged value	Has the tagged value
protocol	Realizes the protocol

© Clear View Training 2008 v2.5

Provided interface syntax

- A provided interface indicates that a classifier implements the services defined in an interface



"Class" style notation

"Lollipop" style notation (note: you can't show the interface operations or attributes with this shorthand style of notation)

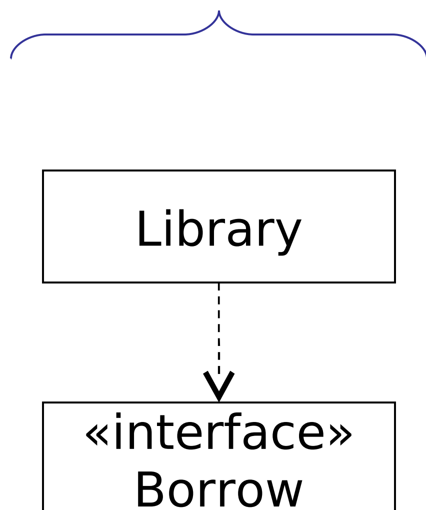
© Clear View Training 2008 v2.5

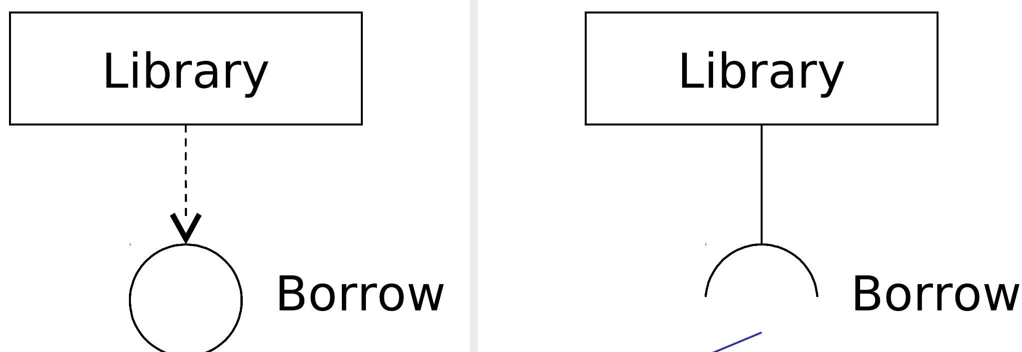
Required interface syntax

- A required interface indicates that a classifier uses the services defined by the interface

class style notation

lollipop style notation

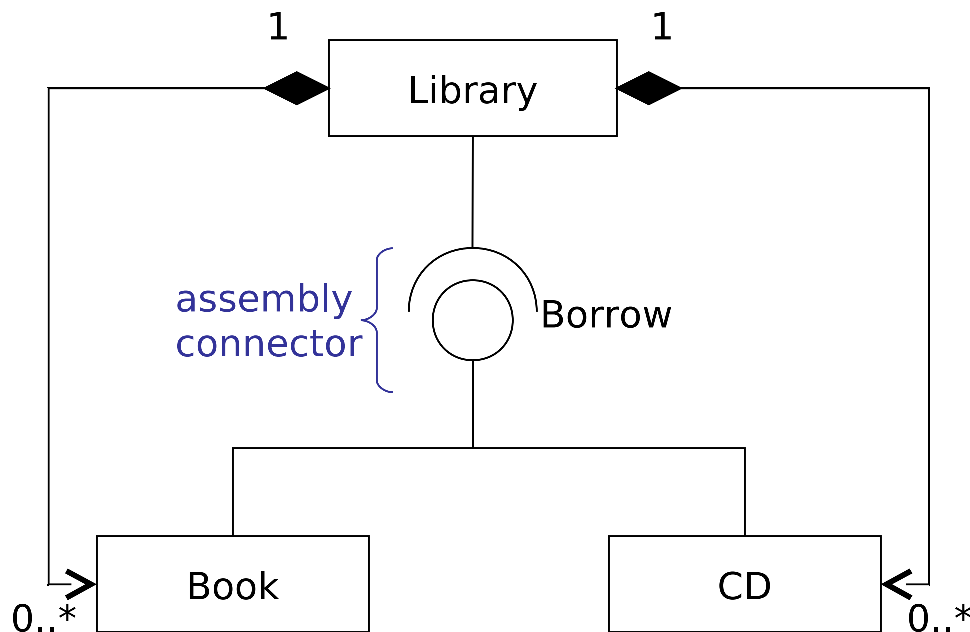




required interface

Assembly connectors

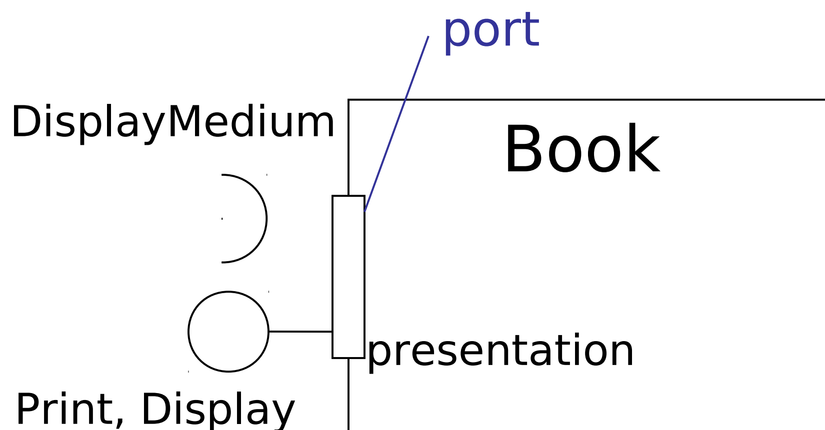
- You can connect provided and required interfaces using an assembly connector

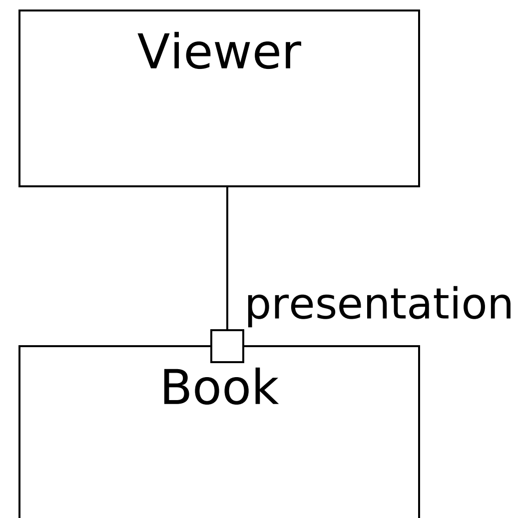


© Clear View Training 2008 v2.5

Ports: organizing interfaces

- A port specifies an interaction point between a classifier and its environment
- A port is typed by its provided and required interfaces:
  - It is a semantically cohesive set of provided and required interfaces
  - It may have a name
- If a port has a single required interface, this defines the type of the port
  - You can name the port portName:RequiredInterfaceName





Interfaces and CBD

- Interfaces are the key to component based development (CBD)
- This is constructing software from replaceable, plug-in parts:
  - Plug – the provided interface
  - Socket – the required interface
- Consider:
  - Electrical outlets
  - Computer ports – USB, serial, parallel
- Interfaces define a contract so classifiers that realise the interface agree to abide by the contract and can be used interchangeably

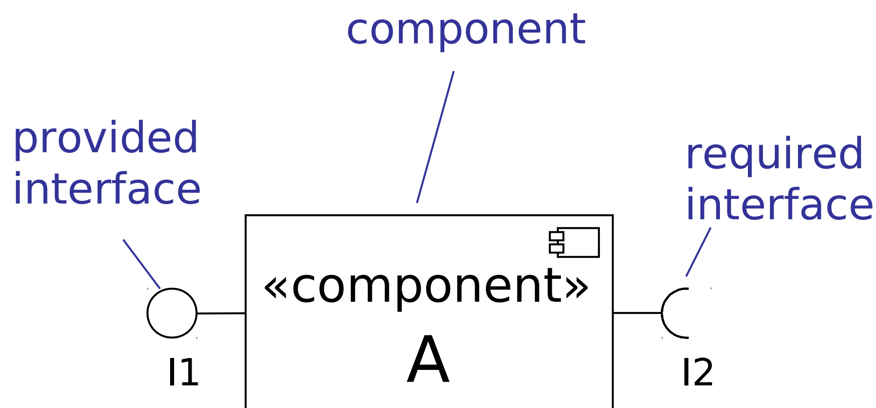
What is a component?

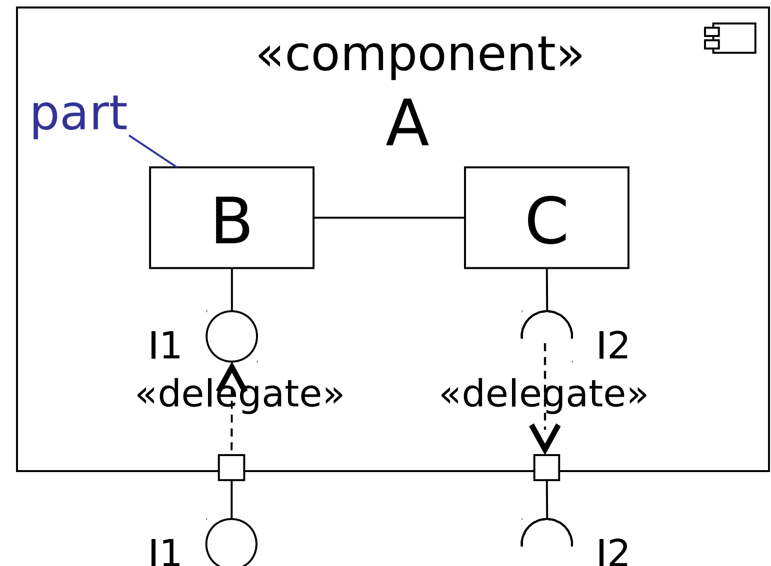
- The UML 2.0 specification states that, "A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment"
  - A black-box whose external behaviour is completely defined by its provided and required interfaces
  - May be substituted for by other components provided they all support the same protocol
- Components can be:
  - Physical - can be directly instantiated at run-time e.g. an Enterprise JavaBean (EJB)
  - Logical - a purely logical construct e.g. a subsystem
    - only instantiated indirectly by virtue of its parts being instantiated

© Clear View Training 2008 v2.5

Component syntax

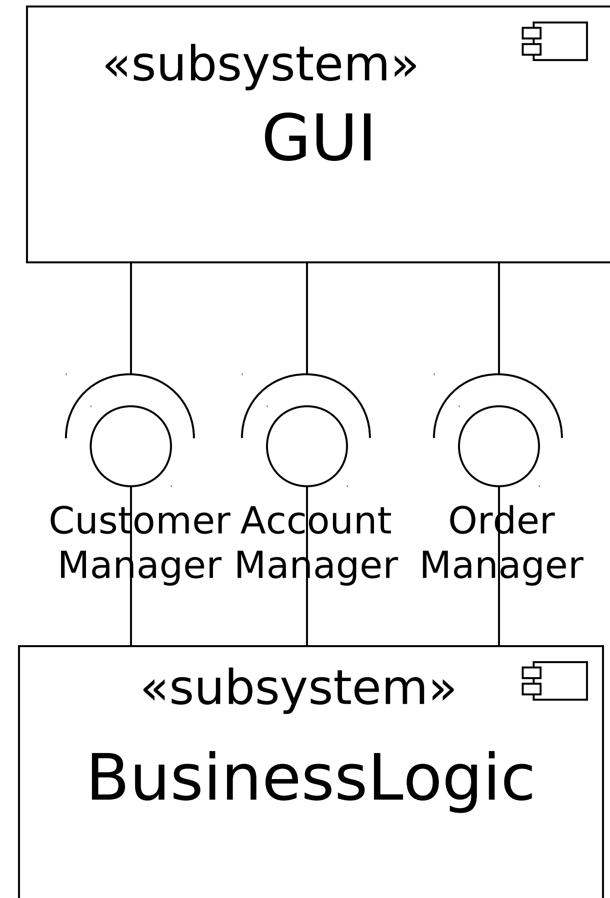
- Components may have provided and required interfaces, ports, internal structure
  - Provided and required interfaces usually delegate to internal parts
  - You can show the parts nested inside the component icon or externally, connected to it by dependency relationships





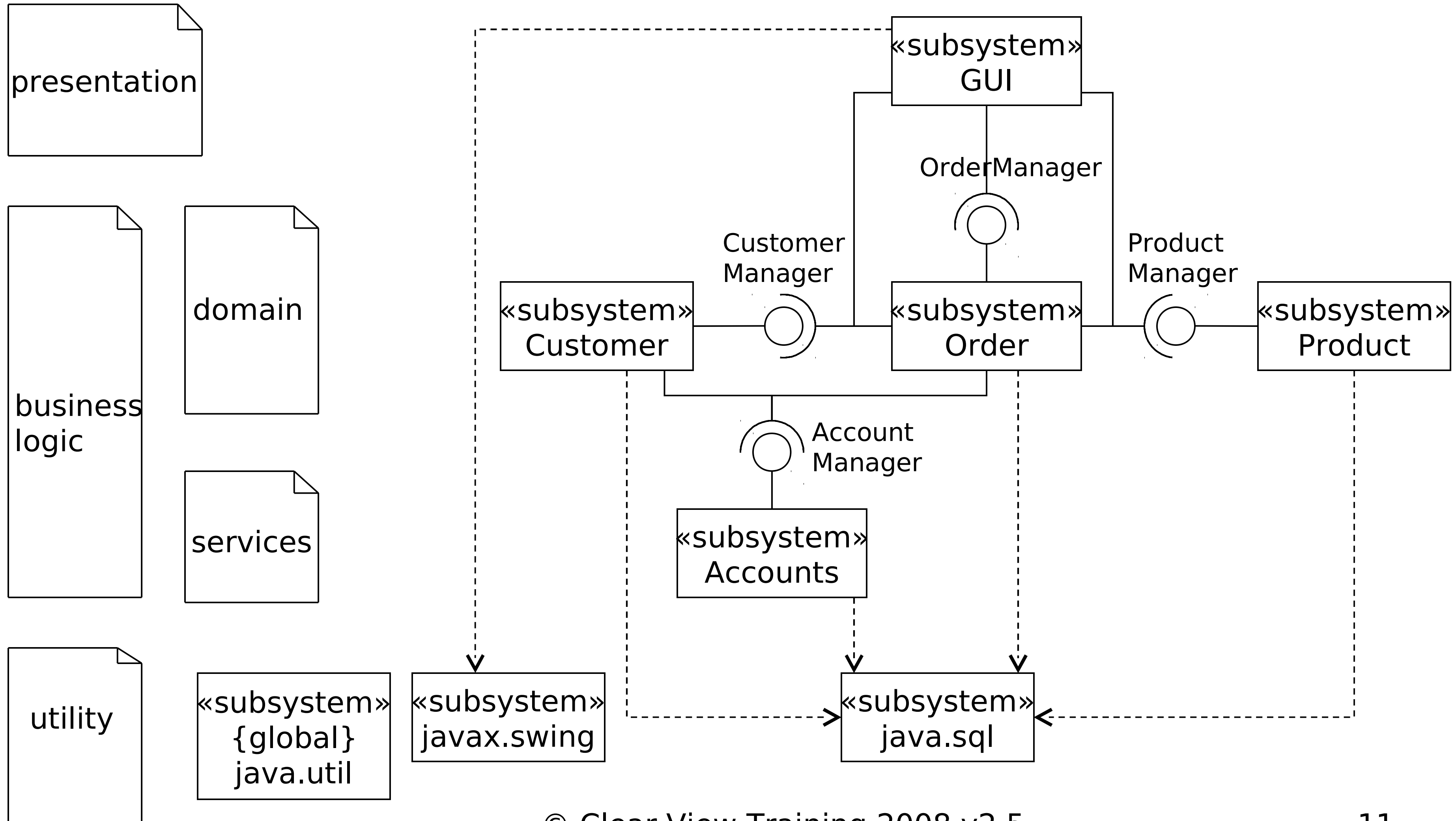
Subsystems

- A subsystem is a component that acts as a unit of decomposition for a larger system
- It is a logical construct used to decompose a larger system into manageable chunks
- Subsystems *can't* be instantiated at run-time, but their contents can
- Interfaces connect subsystems together to create a system architecture



19.12.2

Example layered architecture



Using interfaces

- Advantages:
  - When we design with classes, we are designing to specific implementations
  - When we design with interfaces, we are instead designing to contracts which may be realised by many different implementations (classes)
  - Designing to contracts frees our model from implementation dependencies and thereby increases its flexibility and extensibility
- Disadvantages:
  - Interfaces can add flexibility to systems BUT flexibility may lead to complexity
  - Too many interfaces can make a system too flexible!
  - Too many interfaces can make a system hard to understand

Keep it simple!

© Clear View Training 2008 v2.5

Summary

- Interfaces specify a named set of public features:
  - They define a contract that classes and subsystems may realise
  - Programming to interfaces rather than to classes reduces dependencies between the classes and subsystems in our model
  - Programming to interfaces increases flexibility and extensibility
- Design subsystems and interfaces allow us to:
  - Componentize our system
  - Define an architecture

© Clear View Training 2008 v2.5

Design - use case realization

Use case realization - design

- A collaboration of Design objects and classes that realise a use case
- A Design use case realization contains
  - Design object interaction diagrams
  - Links to class diagrams containing the participating Design classes
  - An explanatory text (flow)

same as in Analysis, but now including implementation details

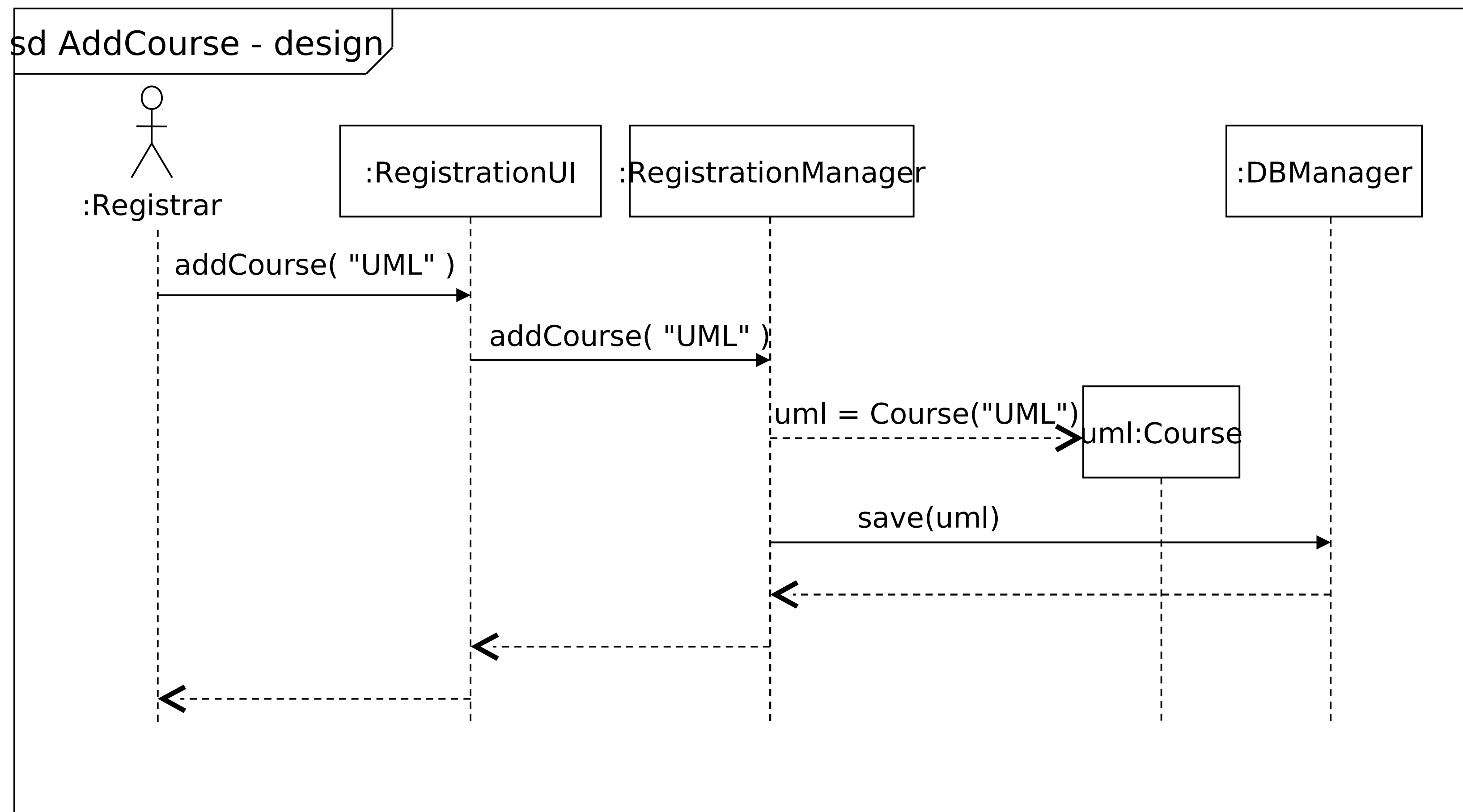
- There is a trace between an Analysis use case realization and a Design use case realization
- The Design use case realization specifies implementation decisions and implements the non-functional requirements

© Clear View Training 2008 v2.5

Interaction diagrams in design

- We only produce a design interaction diagram where it adds value to the project:
  - A refinement of the analysis interaction diagrams to illustrate design issues
  - New diagrams to illustrate technical issues
  - New diagrams to illustrate central mechanisms
- In design:
  - Sequence diagrams are used more than communication diagrams
  - Timing diagrams may be used to capture timing constraints

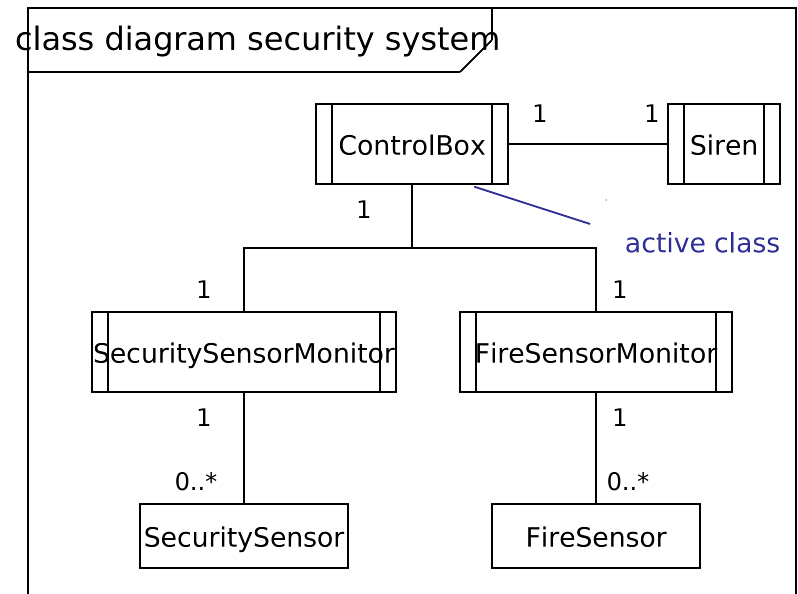
Sequence diagrams in design



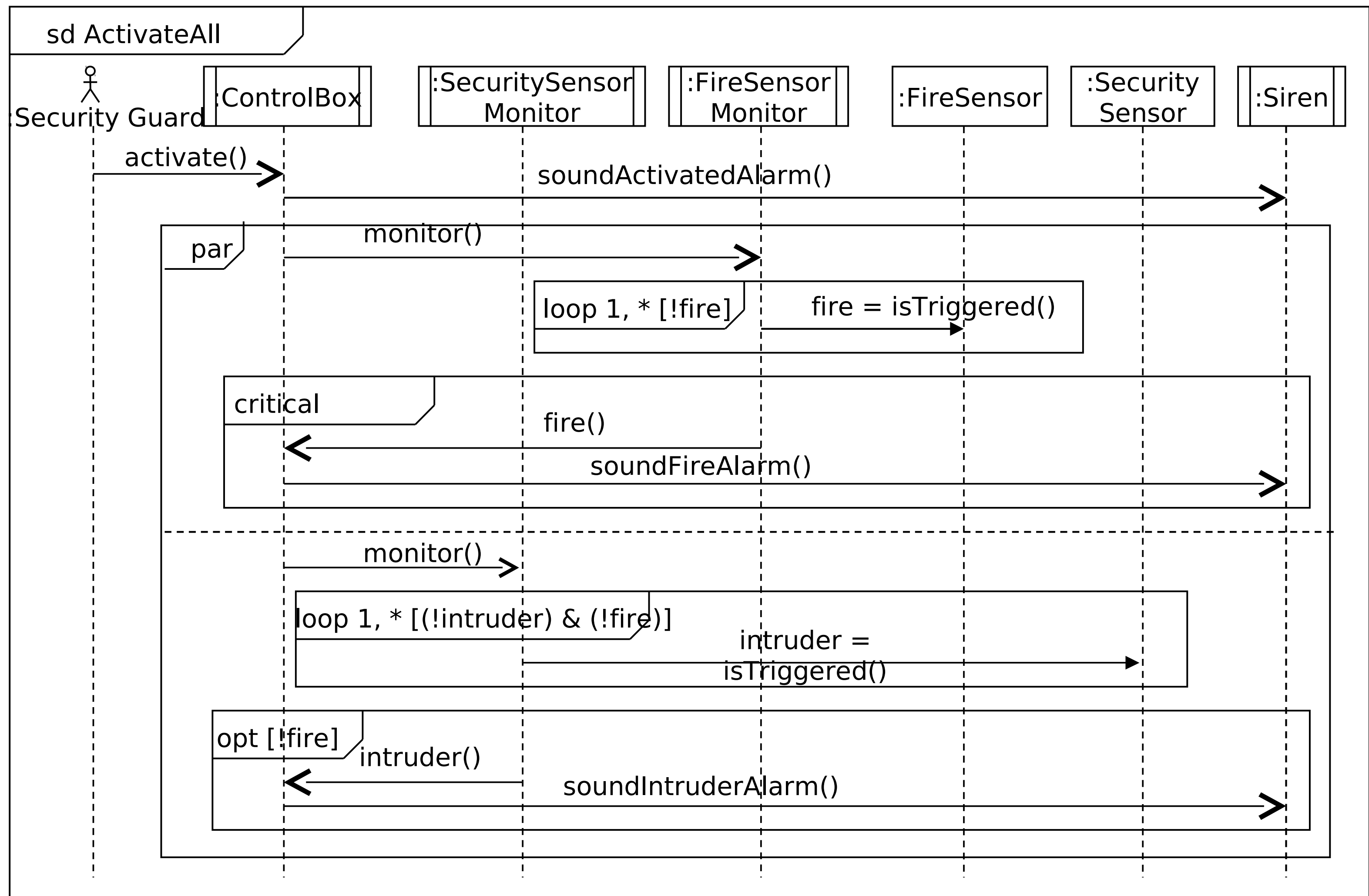
20.5

Concurrency – active classes

- Active classes are classes whose instances are active objects
  - Active objects have concurrent threads of control
- You can show concurrency on sequence diagrams by giving each thread of execution a name and appending this name to the messages (see next slide)

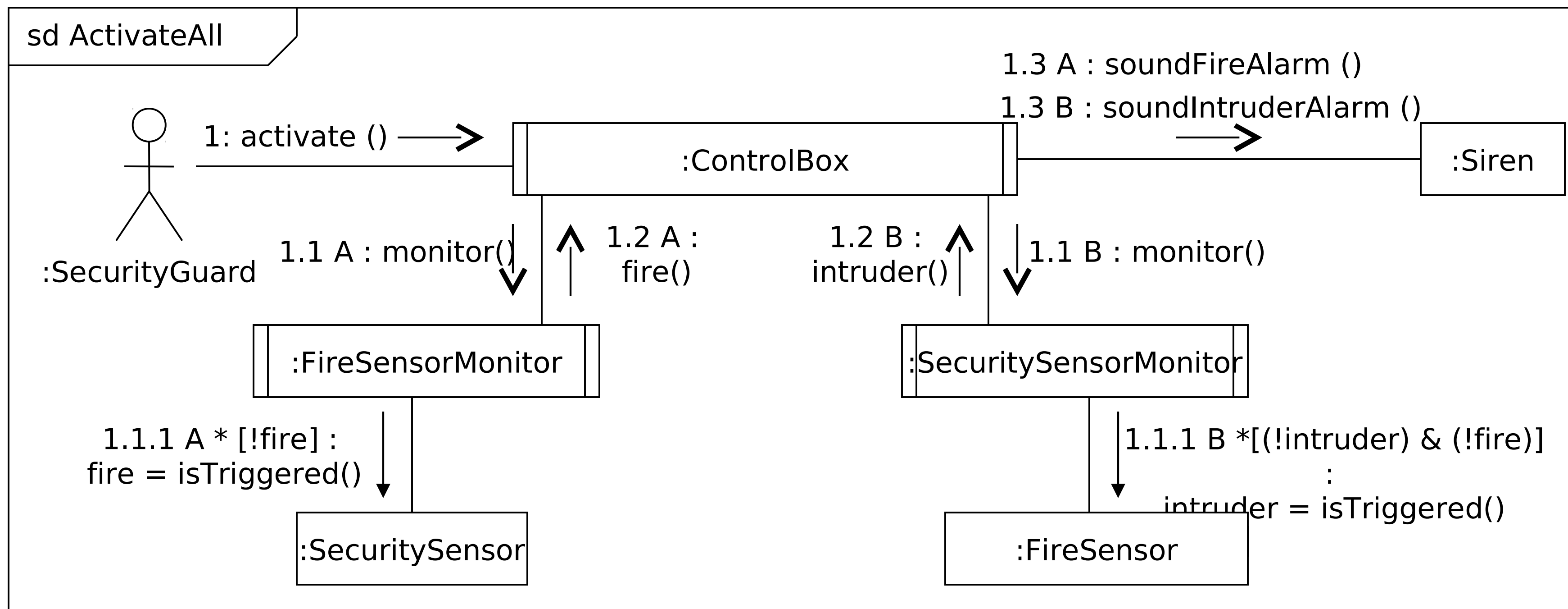


Concurrency with par



© Clear View Training 2008 v2.5

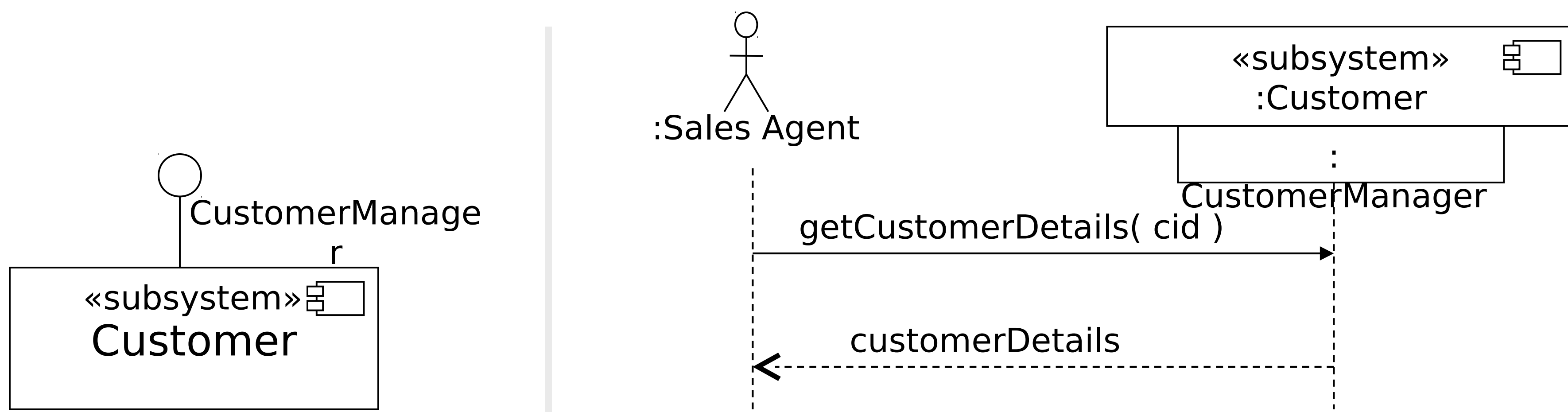
Concurrency – active objects



- Each separate thread of execution is given its own name
  - Messages labelled A execute concurrently to messages labelled B
  - e.g. 1.1 A executes concurrently to 1.1 B

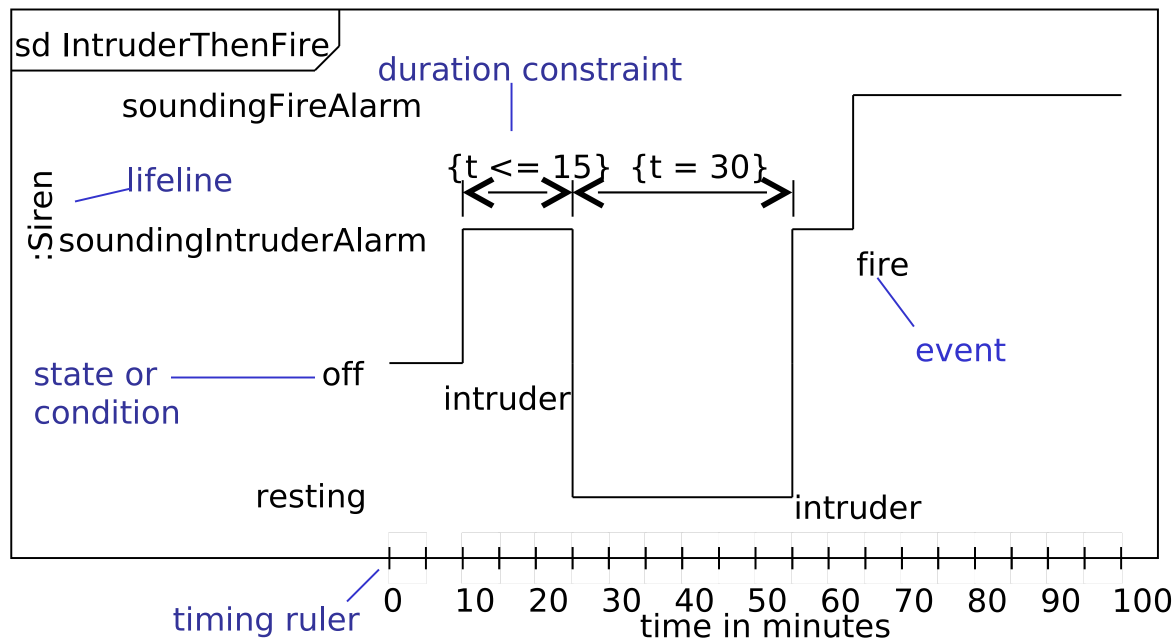
Subsystem interactions

- Sometimes it's useful to model a use case realization as a high-level interaction between subsystems rather than between classes and interfaces
  - Model the interactions of classes within each subsystem in separate interaction diagrams
- You can show interactions with subsystems on sequence diagrams
  - You can show messages going to parts of the subsystem

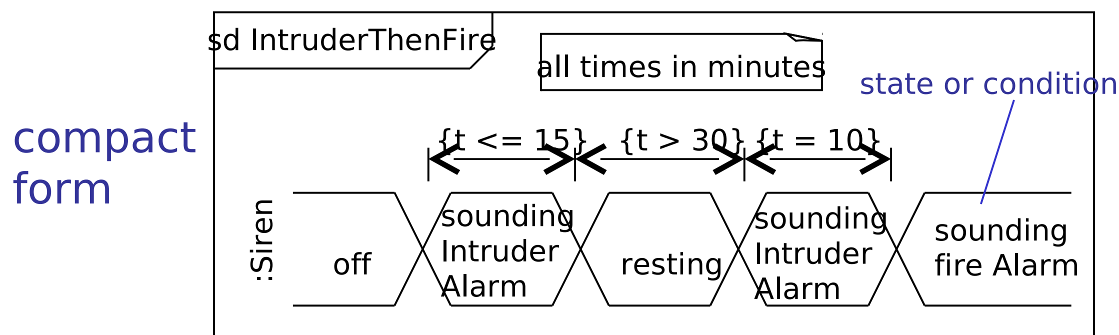


Timing diagrams

- Emphasize the real-time aspects of an interaction
- Used to model timing constraints
- Lifelines, their states or conditions are drawn vertically, time horizontally
- It's important to state the time units you use in the timing diagram

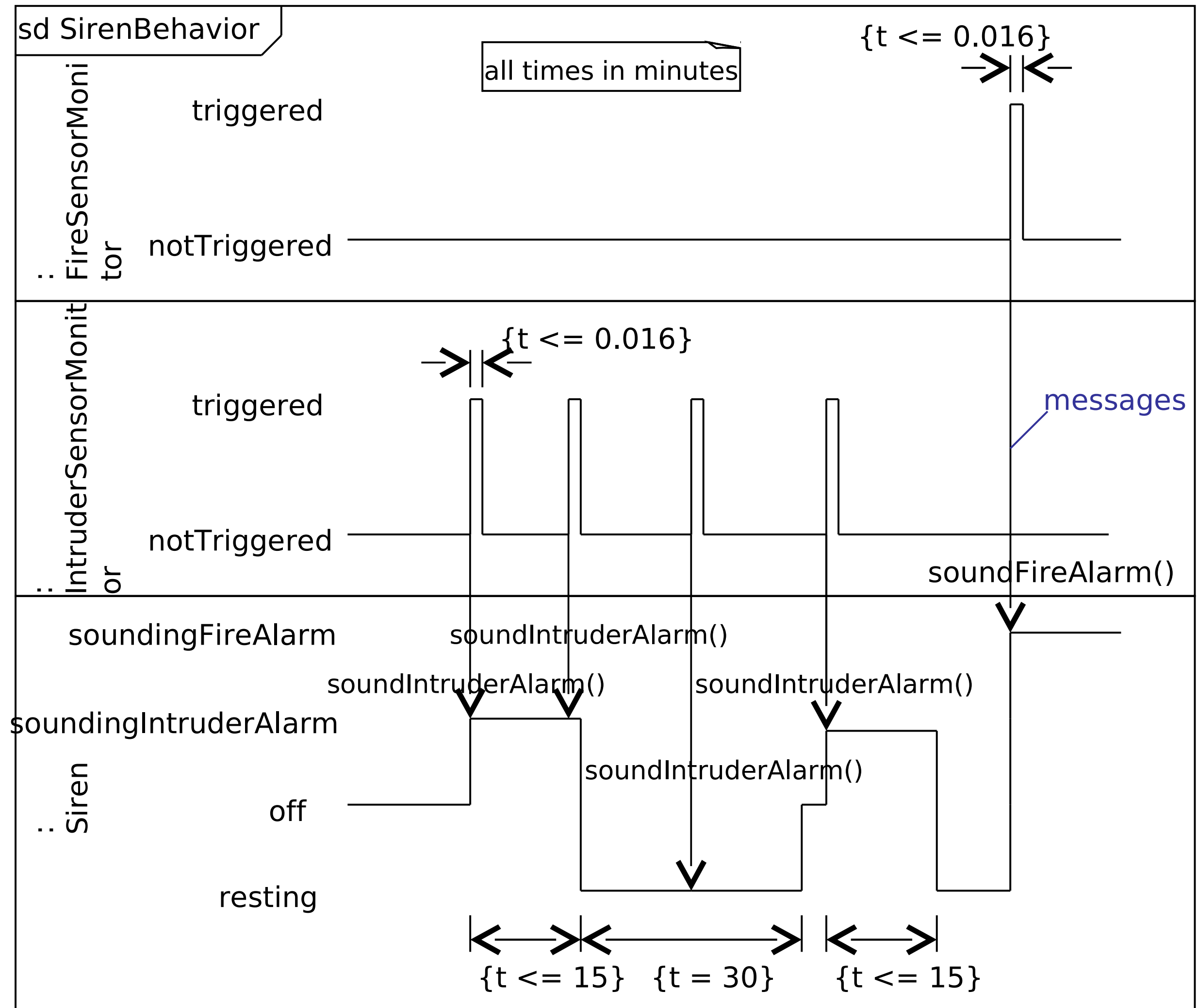


compact form



Messages on timing diagrams

- You can show messages between lifelines on timing diagrams
- Each lifeline has its own partition



Summary

- We have looked at:
  - Design sequence diagrams
  - Subsystem interactions
  - Timing diagrams

Summary

- We have explored advanced aspects of state machines including:
  - Simple composite states
  - Orthogonal composite states
  - Submachine communication
    - Attribute values
  - Submachine states