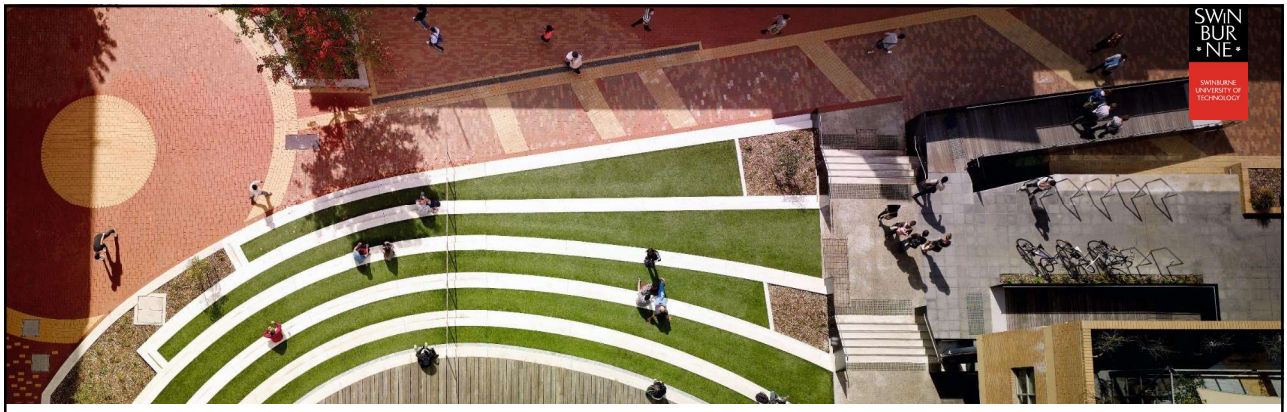


COS80022 – Software Quality and Testing

Week 3 – Modelling and Design

Outline

- Requirements vs. design
- Modelling with UML

Solution Domain

Requirements vs. Design

- Requirements analysis encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product or project, taking account of the possibly conflicting requirements of the various stakeholders, analyzing, documenting, validating and managing software or system requirements
- Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints.

Requirements vs. Design

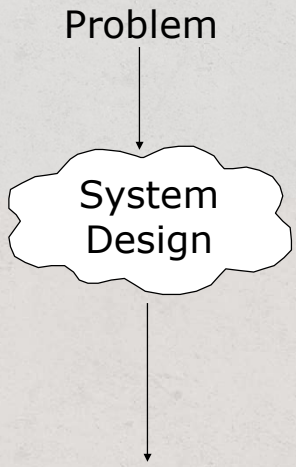
- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable
  - A system architecture may be designed to structure the requirements;
  - The system may inter-operate with other systems that generate design requirements;
  - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
  - This may be the consequence of a regulatory requirement.

Requirements vs. Design

- Analysis: Focuses on the application domain
- Design: Focuses on the solution domain
  - The solution domain is changing very rapidly
    - Halftime knowledge in software engineering: About 3-5 years
    - Cost of hardware rapidly sinking
  - Design knowledge is a moving target
- Design window: Time in which design decisions have to be made.

Scope of System Design

- Bridge the gap
  - between a problem and an existing system in a manageable way
- How? Use Divide & Conquer:
  1. Identify design goals
  2. Model the new system design as a set of subsystems
  3. Address the major design goals.

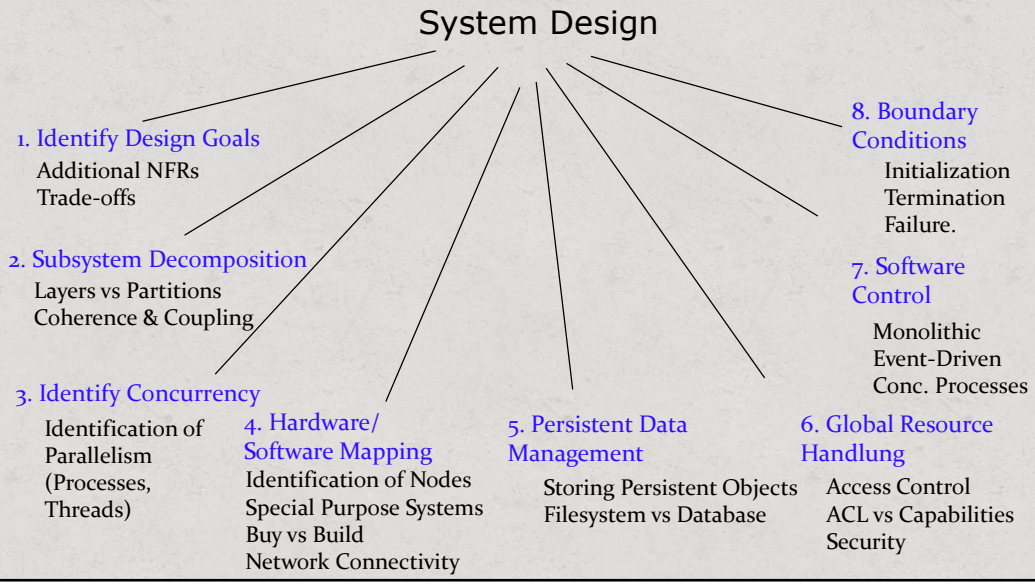


Design

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

- C.A.R. Hoare

System Design: Eight Issues



UML

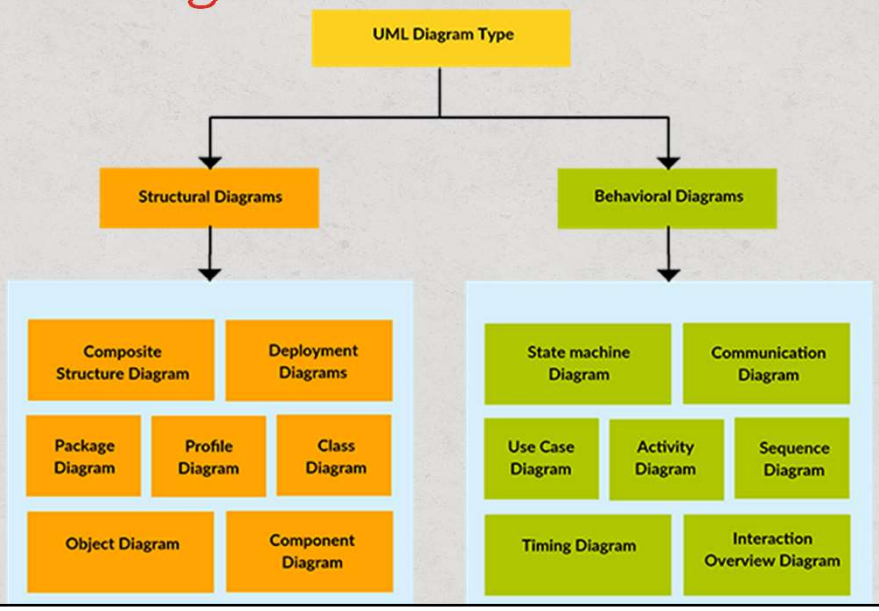
Unified Modelling Language

- Notations enable us to articulate complex ideas succinctly and precisely
- Well-defined semantics
- Well suited for representing a given aspect of a system
- Well understood among participants

Unified Modelling Language

- Various sources
- Standard notation
- Object-oriented methods
- New concepts
- Broad range of applications

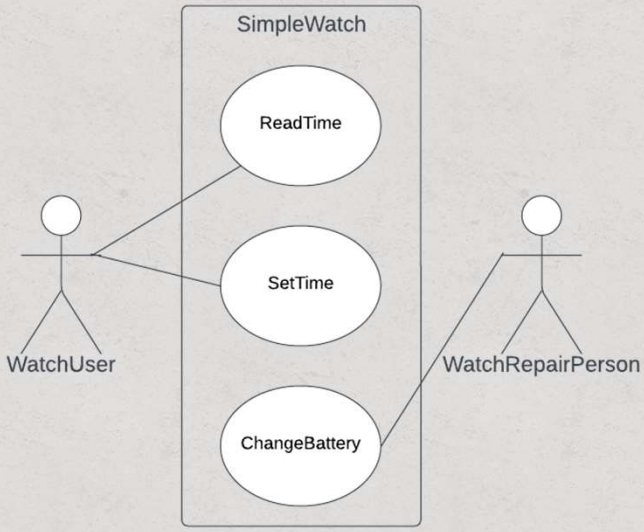
UML Diagrams



UML Diagrams

- Use case diagram
- Component diagram
- Class diagram
- State machine diagram
- Activity diagram

Use Case Diagram



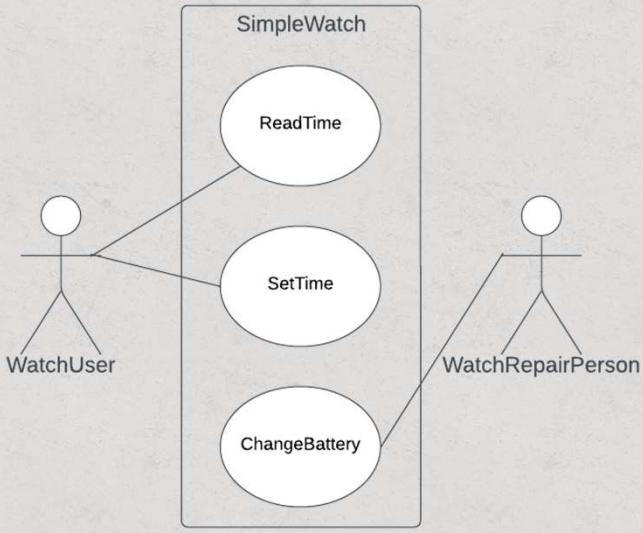
Use Case Diagram

- Use cases: represent functionality
  - Behaviour of the system
  - From external point of view
- Actor: an entity
  - Interact with system
  - Boundary of system

Use Case Diagram

- Fairly simple overview of an interaction
- Lines without arrows
- Need to provide detail
  - Use case description
  - Sequence diagram

Use Case Diagram



Use Case Diagram

- Identifying actors
- Identifying scenarios
- Identifying use cases
- Refining use cases
- Identifying relationships among actors and use cases

Identifying Actors

- External entities
- Human or system
- Role abstractions

Identifying Actors

- Which user groups are supported by the system to perform their work?
- Which user groups execute the system's main functions?
- Which user groups perform secondary functions, such as maintenance and administration?
- With what external hardware or software system will the system interact?

Identifying Scenarios

- Narrative description
- Concrete, focused, informal
- Types
  - As-is
  - Visionary
  - Evaluation
  - Training

Identifying Scenarios

Scenario name	`warehouseOnFire`
Participating actor instances	`bob, alice:FieldOfficer` `john:Dispatcher`
Flow of events	1. Bob, driving down main street in his patrol car, notices smoke coming out of a warehouse. His partner, Alice, activates the "Report Emergency" function from her `FRIEND` laptop. 2. Alice enters the address of the building, a brief description of its location (i.e., northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene, given that the area appears to be relatively busy. She confirms her input and waits for an acknowledgment. 3. John, the `Dispatcher`, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the `Incident` site and sends their estimated arrival time (ETA) to Alice. 4. Alice receives the acknowledgment and the ETA.

Identifying Scenarios

- What are the tasks that the actor wants the system to perform?
- What information does the actor access? Who creates that data? Can it be modified or removed? By whom?
- Which external changes does the actor need to inform the system about? How often? When?
- Which events does the system need to inform the actor about? With what latency?

Identifying Use Cases

- Scenario vs. use case
  - Scenario: instance of use case
  - Use case: all possible scenarios for functionality
- Complete flow of events
  - Initiated by actor
  - Interact with other actors

Identifying Use Cases

Use case name	`ReportEmergency`
Participating actors	Initiated by `FieldOfficer` Communicates with `Dispatcher`
Flow of events	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. `FRIEND` responds by presenting a form to the `FieldOfficer`. 3. The `FieldOfficer` completes the form by selecting the emergency level, type, location, and brief description of the situation. The `FieldOfficer` also describes possible responses to the emergency situation. Once the form is completed, the `FieldOfficer` submits the form. 4. `FRIEND` receives the form and notifies the `Dispatcher`. 5. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. The `Dispatcher` selects a response and acknowledges the report. 6. `FRIEND` displays the acknowledgment and the selected response to the `FieldOfficer`.
Entry condition	• The `FieldOfficer` is logged into `FRIEND`.
Exit conditions	• The `FieldOfficer` has received an acknowledgment and the selected response from the `Dispatcher`, OR • The `FieldOfficer` has received an explanation indicating why the transaction could not be processed.
Quality requirements	• The `FieldOfficer`'s report is acknowledged within 30 seconds. • The selected response arrives no later than 30 seconds after it is sent by the `Dispatcher`.

Identifying Use Cases

- Use cases: named with verb phrases
- Actors: named with noun phrases
- Clear boundary
- Use case steps: in active voice
- Clear relationship between steps
- Separate description of exceptions
- Should not describe user interface
- Not too long

Identifying Use Cases – A Bad Example

Use case name	`Accident`	*Bad name: What is the user trying to accomplish?*
Initiating actor	Initiated by `FieldOfficer`	
Flow of events	1. The `FieldOfficer` reports the accident.	
	2. An ambulance is dispatched.	*Causality: Which action caused the `FieldOfficer` to receive an acknowledgment?*
		Passive voice: Who dispatches the ambulance?
	3. The `Dispatcher` is notified when the ambulance arrives on site.	*Incomplete transaction: What does the `FieldOfficer` do after the ambulance is dispatched?*

Refining Use Cases

- Define functionality for early validation
- Cost of changing requirements increases with development
  - Address most requirements issues early
- Exploration work: scenarios and mock-ups

Refining Use Cases

Use case name	`ReportEmergency`
Participating actors	Initiated by `FieldOfficer` Communicates with `Dispatcher`
Flow of events	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. `FRIEND` responds by presenting a form to the officer. *The form includes an emergency type menu (general emergency, fire, transportation) and location, incident description, resource request, and hazardous material fields.* 3. The `FieldOfficer` completes the form by *specifying minimally the emergency type and description fields*. The `FieldOfficer` may also describe possible responses to the emergency situation *and request specific resources*. Once the form is completed, the `FieldOfficer` submits the form. 4. `FRIEND` receives the form and notifies the `Dispatcher` *by a pop-up dialog*. 5. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. *All the information contained in the `FieldOfficer`'s form is automatically included in the `Incident`. The `Dispatcher` selects a response by allocating resources to the `Incident` (with the `AllocateResources` use case) and acknowledges the emergency report by sending a short message to the `FieldOfficer`.* 6. `FRIEND` displays the acknowledgment and the selected response to the `FieldOfficer`.
Entry condition	• ...

Refining Use Cases

- Use scenarios for communications
- Refine a single scenario
- Define the scope of system
- Mock-ups as visual support
- Multiple alternatives
- Broad vertical slice

Refining Use Cases

- Completeness and correctness
- Functionality not covered by scenarios
- Seldom occurring cases
- Exception handling
- More details about features
- Constraints

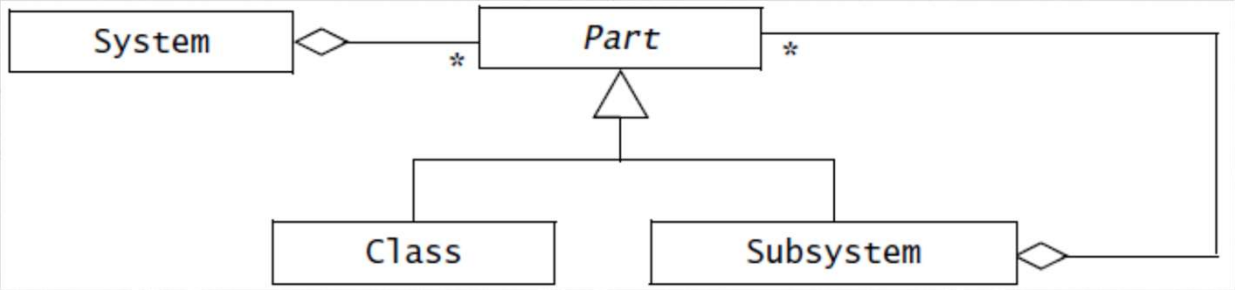
Use Cases for Testing

- Acceptance testing
- System testing

Subsystem and Class

- To reduce the complexity of the application domain, we identified smaller parts called “classes” and organized them into packages.
- To reduce the complexity of the solution domain, we decompose a system into simpler parts, called “subsystems,” which are made of a number of solution domain classes

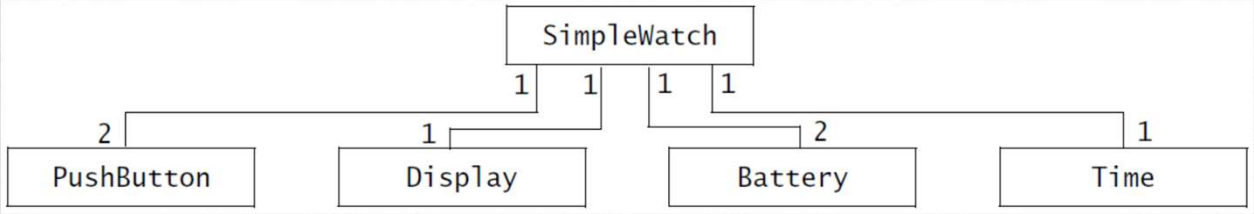
Subsystem and Class



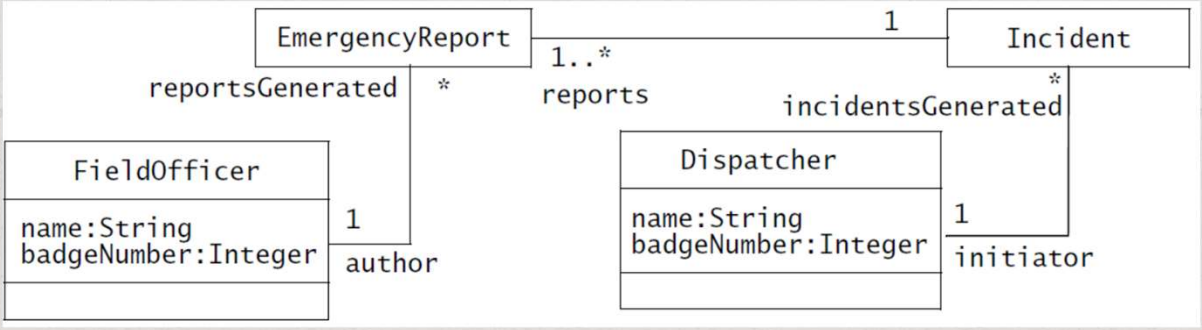
Class Diagram

- Class: abstraction specifying common structure and behaviour of a set of objects
- Object: instance of class
- Class diagram: Describe the structure of system
  - In terms of objects, classes, attributes, operations, and their associations.

Class Diagram



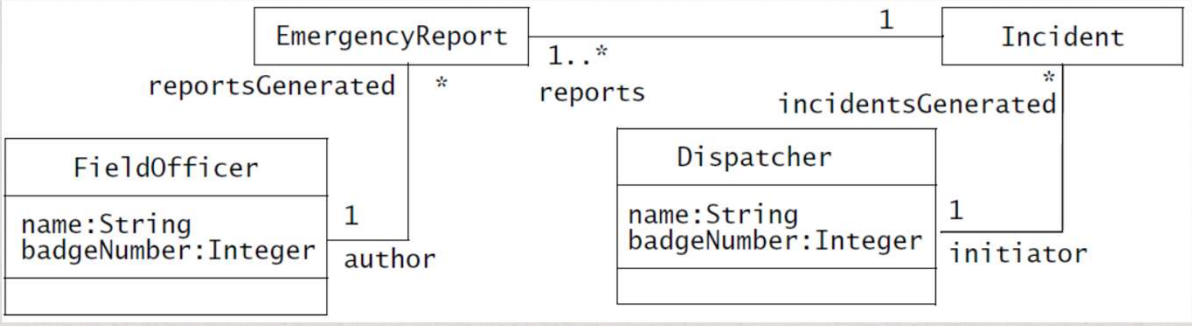
Class Diagram



Association

- Relationships between classes

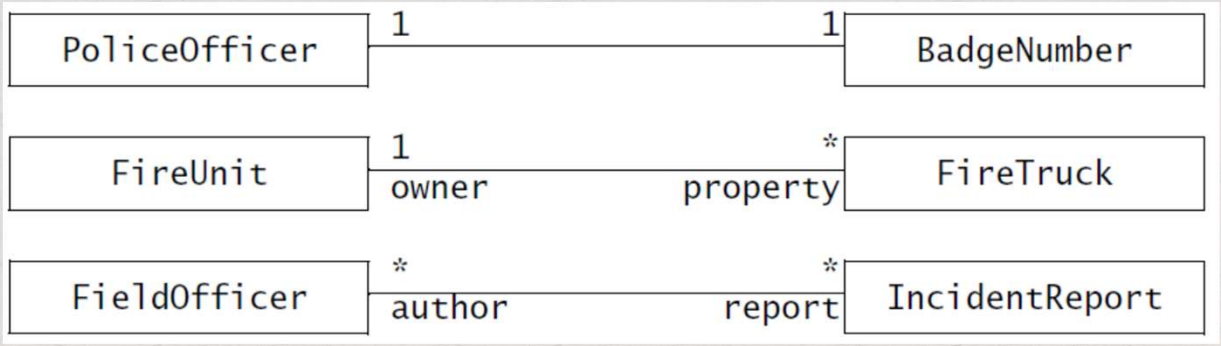
Roles



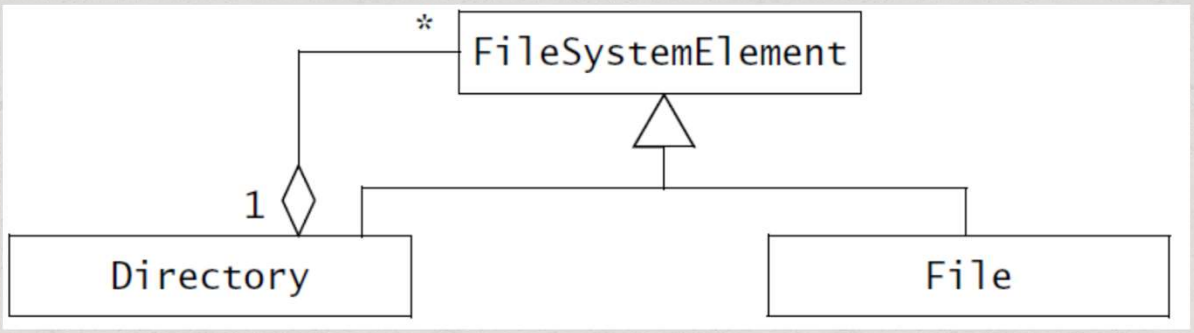
Multiplicity

- Number of links that can legitimately originate from an instance of class
  - One-to-one association
  - One-to-many association
  - Many-to-many association

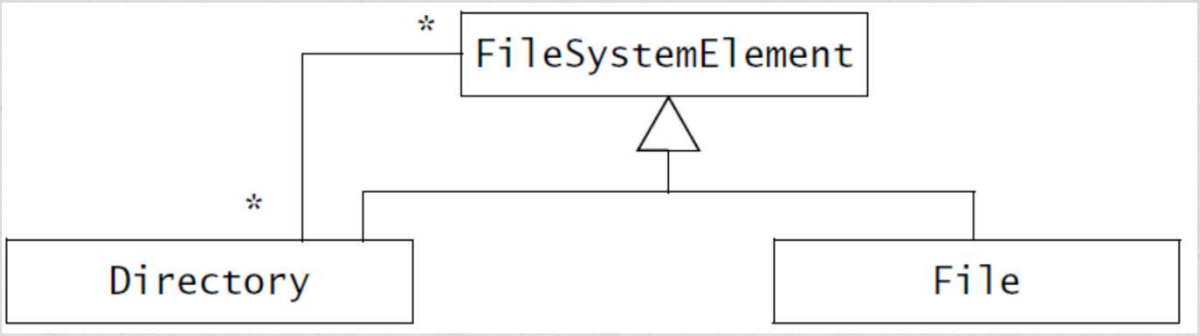
Multiplicity



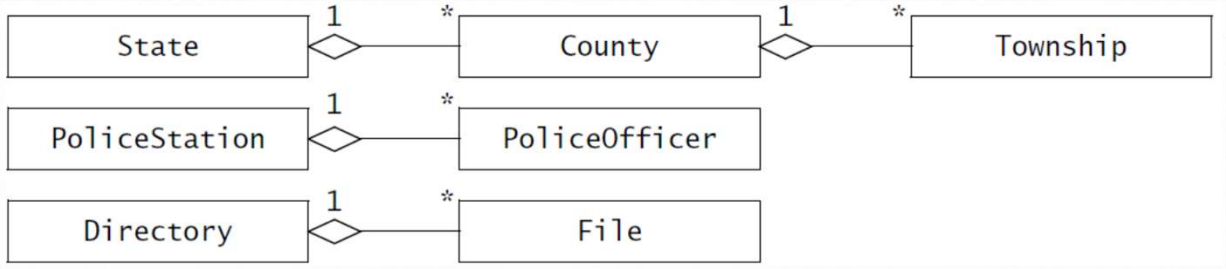
Multiplicity



Multiplicity

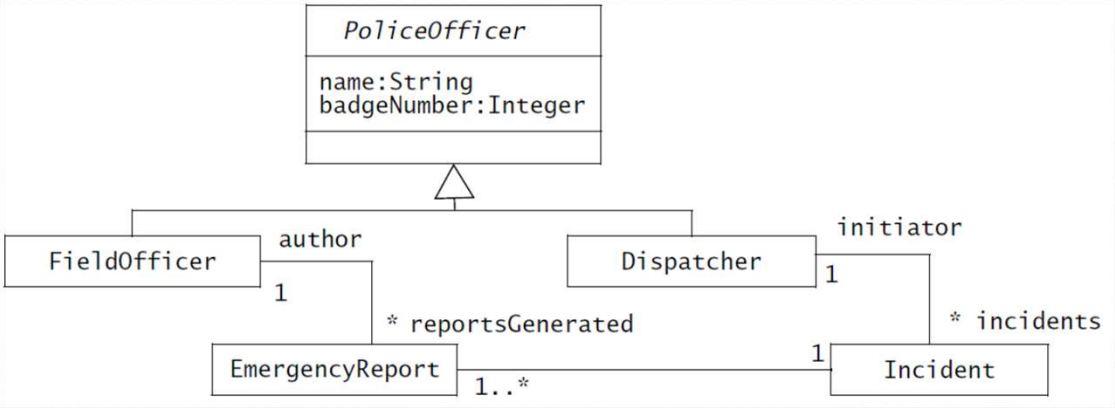


Aggregation



Inheritance

- Relationship between a general class and one or more specialised classes



Class Diagram for Testing

- System testing
- Integration testing
- Black-box testing

Subsystem

- A replaceable part of the system
- Typically corresponds to the amount of work tackled by a single developer or a single development team
- Concurrent teams can work on individual subsystems with minimal communication overhead
- Recursive decomposition

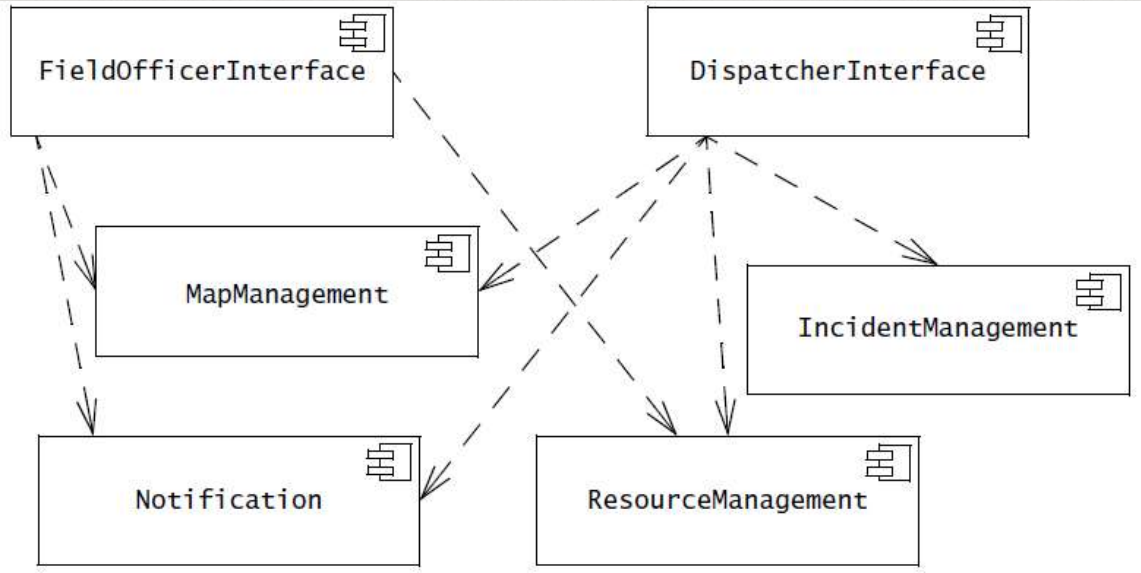
Subsystem and Component

- Subsystems can be depicted as components
- A logical component corresponds to a subsystem that has no explicit run-time equivalent
  - e.g. individual business components that are composed together into a single run-time application logic layer
- A physical component corresponds to a subsystem that as an explicit run-time equivalent
  - e.g., a database server

Component Diagram

- Depicts how components are wired together to form larger components or software systems
- Illustrate the structure of arbitrarily complex systems
- Components are depicted as rectangles with the component icon in the upper right corner
- Dependencies among components can be depicted with dashed stick arrows

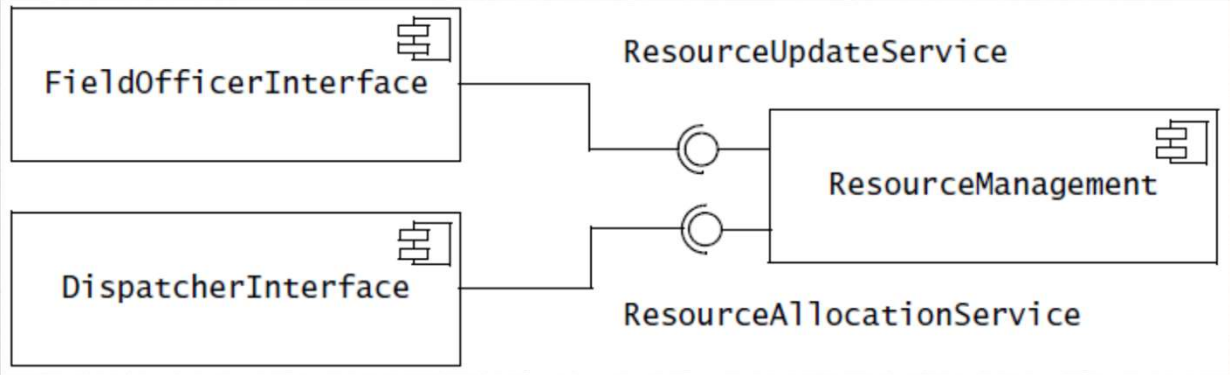
Component Diagram



Component Diagram

- The set of operations of a subsystem that are available to other subsystems form the subsystem interface
- Provided and required interfaces can be depicted in UML with assembly connectors, also called ball-and-socket connectors

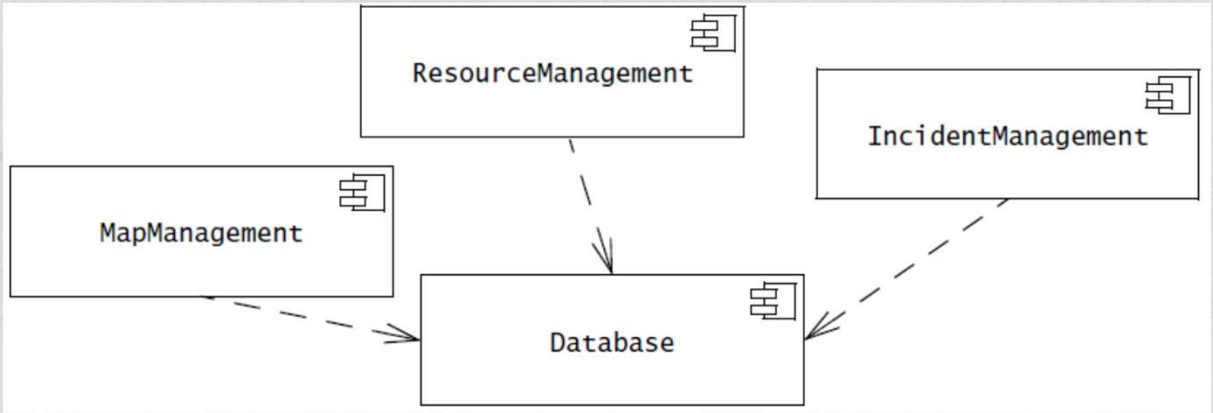
Component Diagram



Component Diagram

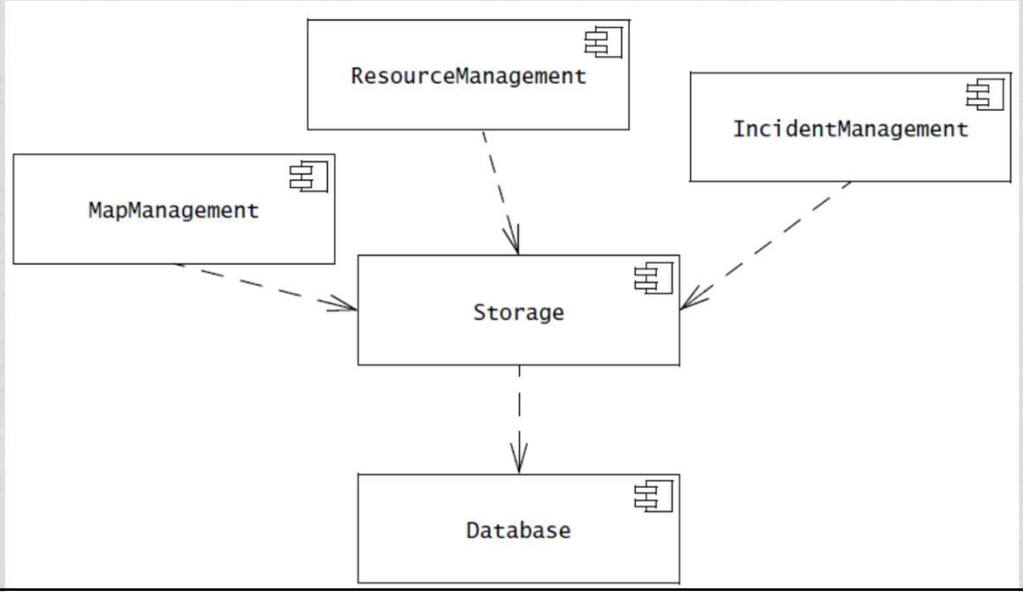
- Coupling: the number of dependencies between two subsystems
- Loose coupling
  - If two subsystems are loosely coupled, they are relatively independent, so modifications to one of the subsystems will have little impact on the other
  - A desirable property of a subsystem decomposition is that subsystems are as loosely coupled as reasonable

Component Diagram



55

Component Diagram

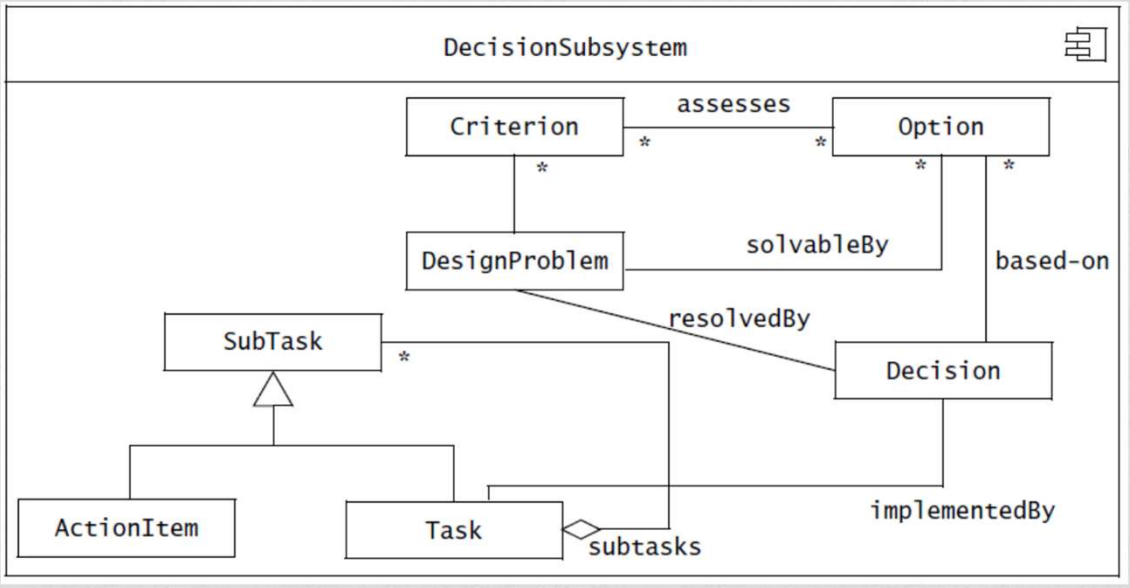


56

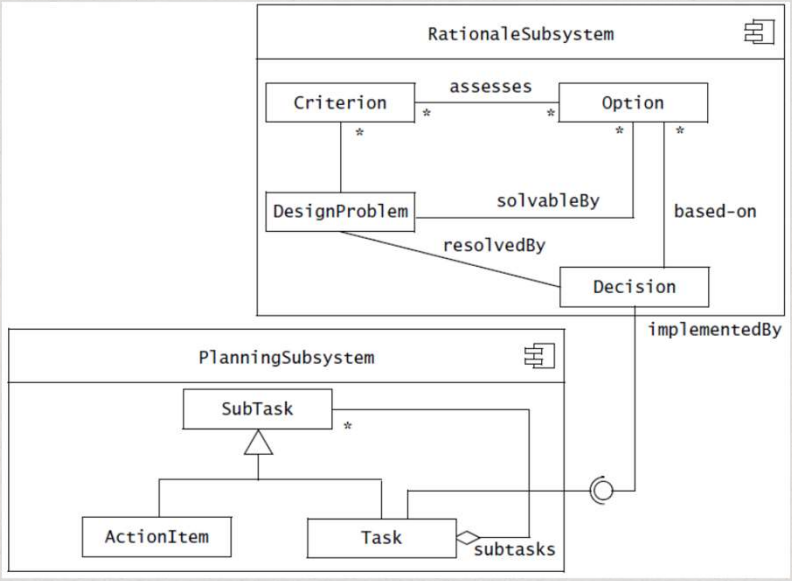
Component Diagram

- Cohesion: the number of dependencies within a subsystem
- High cohesion
  - If a subsystem contains many objects that are related to each other and perform similar tasks, its cohesion is high
  - A desirable property of a subsystem decomposition is that it leads to subsystems with high cohesion

Component Diagram



Component Diagram



Component Diagram

- Trade-off between cohesion and coupling
  - Increase cohesion by decomposing the system into smaller subsystems.
  - This also increases coupling as the number of interfaces increases.
- 7 ± 2 concepts at any level of abstraction.
  - More than nine or more than nine services consider revising the decomposition.
  - The number of layers should not be more than 7 ± 2. Preferably, just three layers.

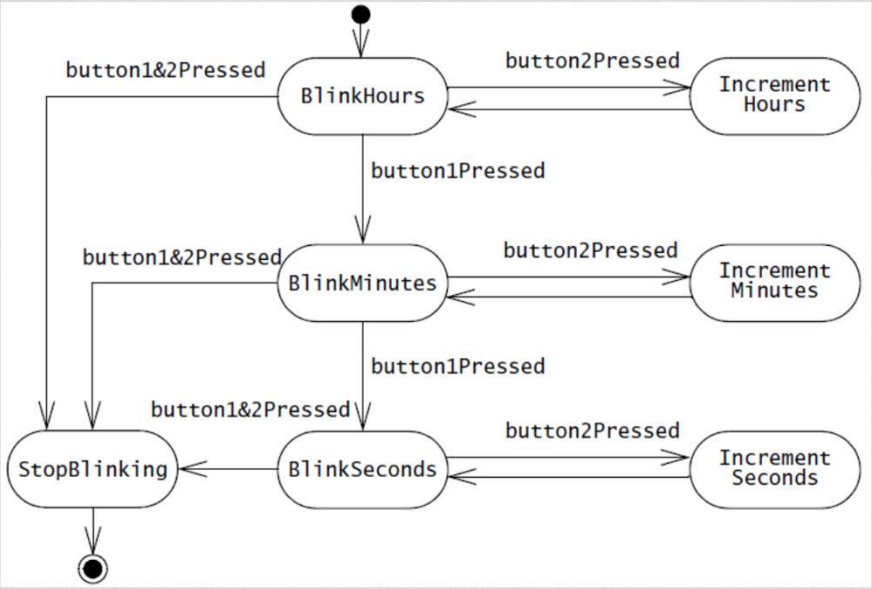
Component Diagram for Testing

- System testing
- Integration testing
- Black-box testing

State Machine Diagram

- Describe the dynamic behavior of an individual object as a number of states and transitions between these states
- A state represents a particular set of values for an object
- Given a state, a transition represents a future state the object can move to and the conditions associated with the change of state

State Machine Diagram



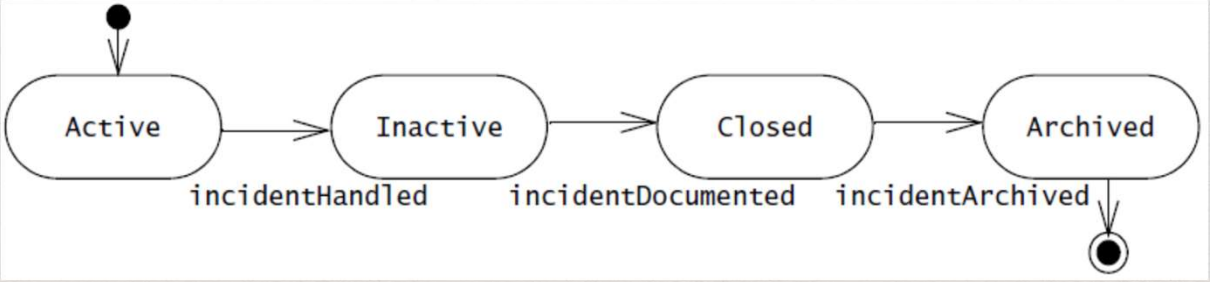
State Machine Diagram

- A UML state machine is a notation for describing the sequence of states an object goes through in response to external events.
- UML state machines are extensions of the finite state machine model.
  - State machines provide notation for nesting states and state machines
  - State machines provide notation for binding transitions with message sends and conditions on objects

State Machine Diagram

- A state is a condition satisfied by the attributes of an object
  - Depicted by a rounded rectangle
  - Labelled with their name
  - Small solid black circle: initial state
  - Circle surrounding a small solid black circle: final state
- A transition represents a change of state triggered by events, conditions, or time
  - Depicted by an open arrow connecting two states

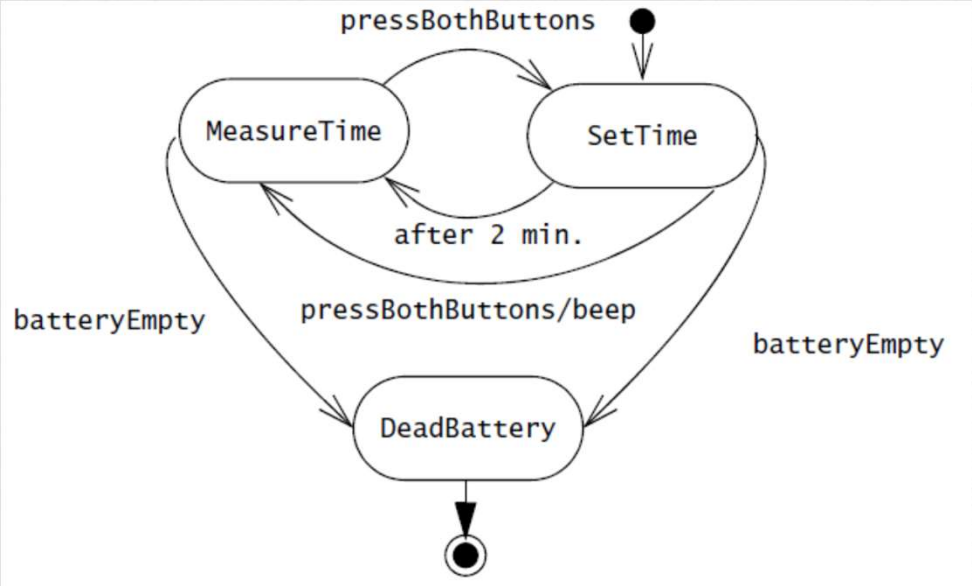
State Machine Diagram



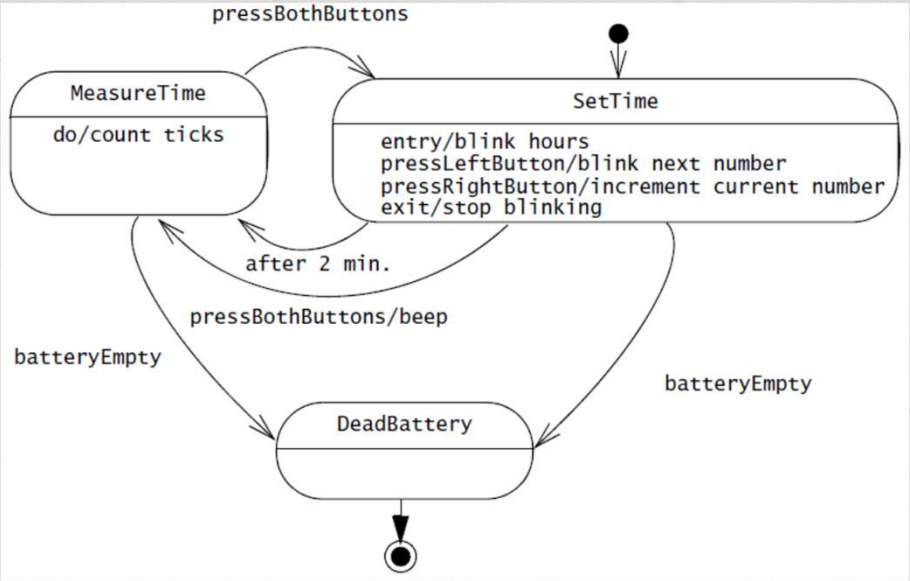
State Machine Diagram

- Actions: fundamental units of processing that can take a set of inputs, produce a set of outputs, and can change the state of the system.
  - Normally take a short amount of time to execute and are not interruptable.
- Internal transition: does not leave the state
  - Triggered by events and can have associated actions
- Activity: a coordinated set of actions
  - Can take long time and can be interrupted
  - Associated with state using the do label

State Machine Diagram



State Machine Diagram



State Machine Diagram

- Represent nontrivial behaviour of a subsystem or an object.
- Make explicit which attribute or set of attributes have an impact on the behaviour of a single object.
- Identify object attributes and refine the behaviour description of an object
- Describe solution domain objects with interesting behaviour

State Machine Diagram

- Represent behaviour from the perspective of a single object
- Enable the developer to build a more formal description of the behaviour of the object, and consequently, to identify missing use cases.
- May identify new behaviour
- Not necessary to build state machines for every class in the system. Only objects with an extended lifespan and state-dependent behaviour are worth considering.

State Machine Diagram for Testing

- Component testing
- Black-box testing

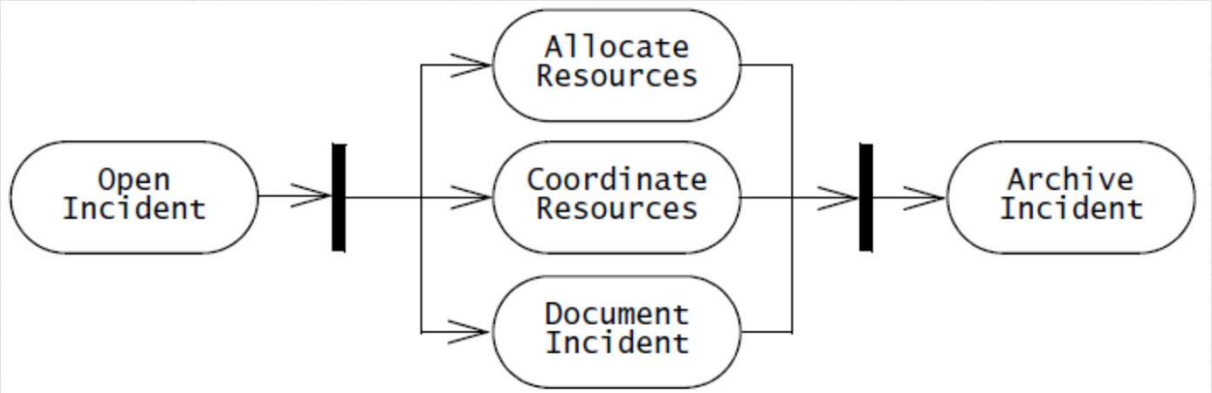
Activity Diagram

- An activity diagram describes the behaviour of a system in terms of activities.
- Activities are modelling elements that represent the execution of a set of operations.
- The execution of an activity can be triggered by the completion of other activities, by the availability of objects, or by external events.

Activity Diagram

- Rounded rectangles represent activities
- Arrows between activities represent control flow
- Thick bars represent the synchronization of the control flow

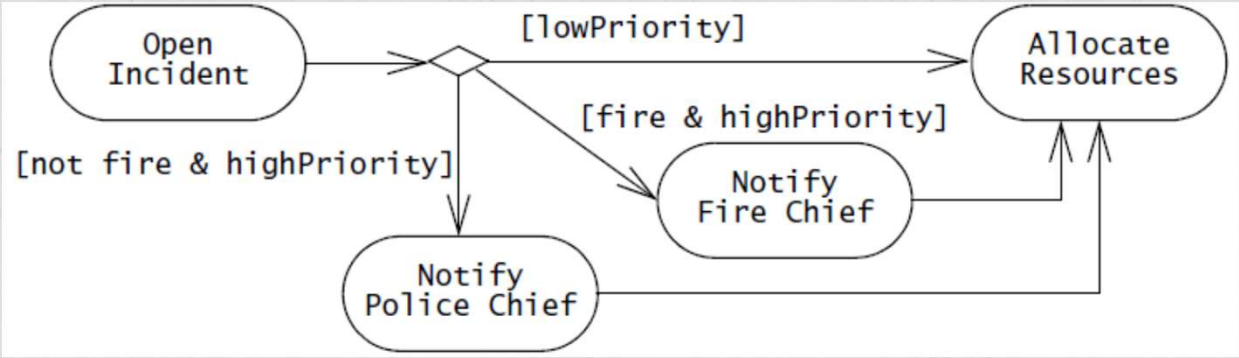
Activity Diagram



Activity Diagram

- **Control nodes**: coordinate control flows in an activity diagram, providing mechanisms for representing decisions, concurrency, and synchronization.
  - Decisions
  - Fork nodes
  - Join nodes

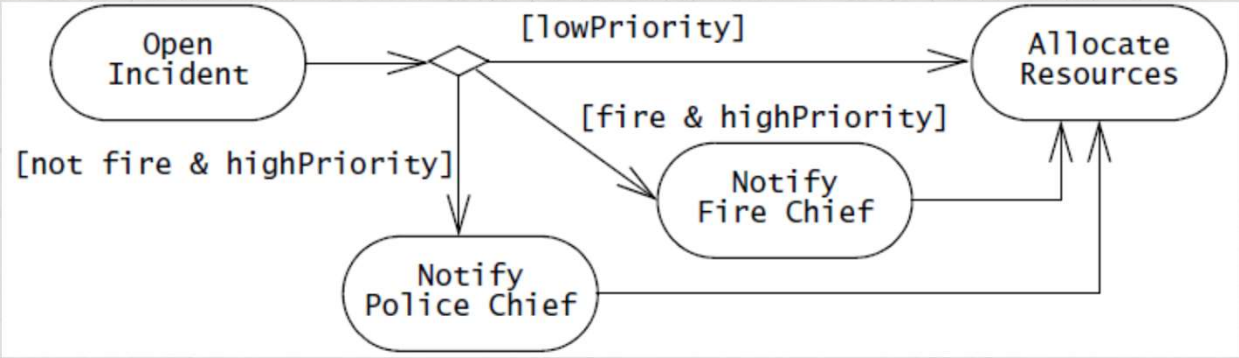
Activity Diagram



Activity Diagram

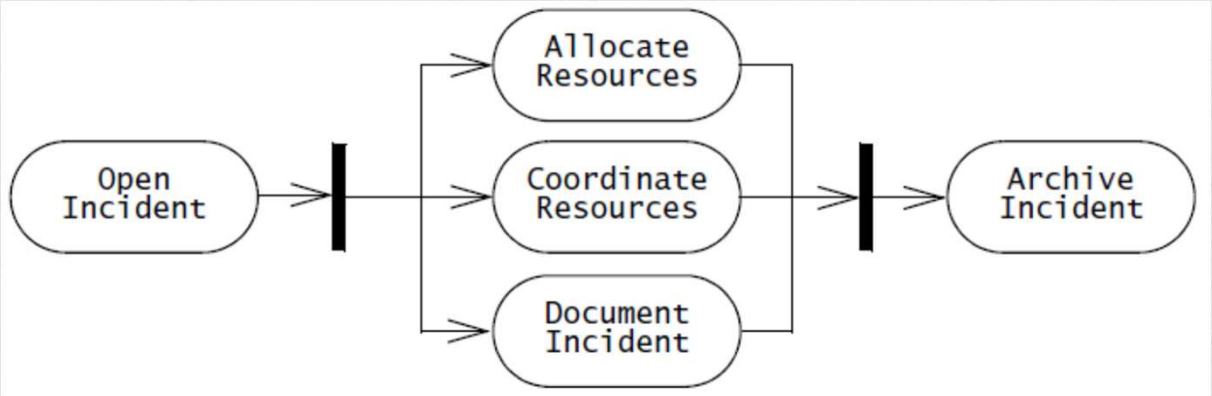
- **Fork nodes** and **join nodes** represent concurrency
  - Fork nodes denote the splitting of the flow of control into multiple threads
  - Join nodes denotes the synchronization of multiple threads and their merging of the flow of control into a single thread

Activity Diagram



79

Activity Diagram

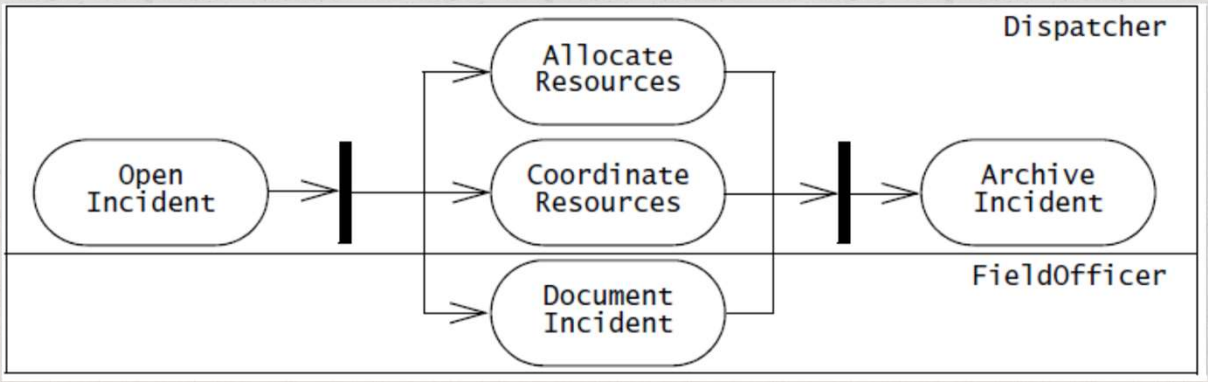


80

Activity Diagram

- **Swimlanes** (also called activity partitions):
  - Activities may be grouped into swimlanes to denote the object or subsystem that implements the actions
  - Represented as rectangles enclosing a group of actions
  - Transitions may cross swimlanes.

Activity Diagram



Activity Diagram

- Provide a task-centric view of the behaviour of a set of objects.
- Describe sequencing constraints among use cases
- Describe sequential activities among a group of objects,
- Describe tasks of a project.

Activity Diagram for Testing

- Component testing
- White-box testing

Summary

- Use case diagram
- Class diagram
- Component diagram
- State machine diagram
- Activity diagram

References

- Sommerville, I. *Software Engineering* (Chapter 5, 6)
- Bruegge, B. & Dutoit A.H. *Object-Oriented Software Engineering Using UML, Patterns, and Java* (Chapter 2)

Next

- Tutorial
  - UML
- Next week
  - Testing foundation