# SWEN30006 Exam Notes 2018 S1

**Software Design**: Purposefully choosing the structure and behaviour of the software system. Behaviour is about how the system responds to inputs and events, and choosing how parts of the system collaborate to achieve goals.
**Software Modelling**: Creation of tangible, but abstract, representations of a system so that design ideas can be communicated and critiqued, and alternatives explored.
**Analysis:** investigation of the problem & requirements
**Object-Oriented Analysis**: Finding & describing objects & concepts in the problem domain
**Design**: conceptual solution to a problem that meets the requirements
**Implementation**: a concrete solution to a problem that meets the requirements

## USE CASES

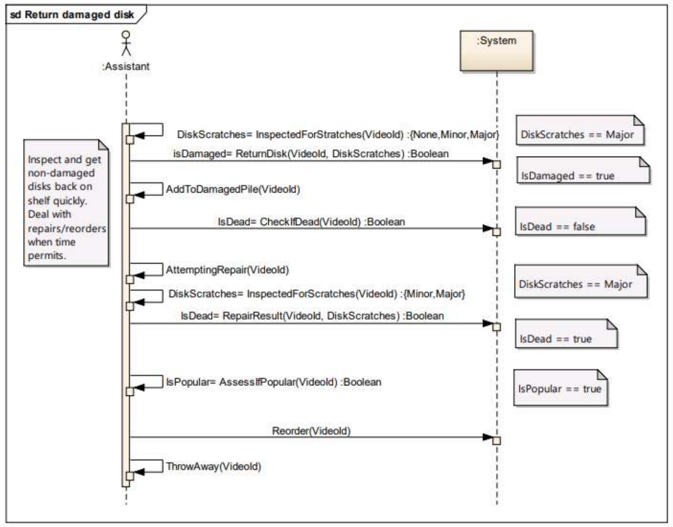
Text descriptions of how particular users or actors interact with the system to achieve a goal. Purpose is to discover and record functional and behavioural requirements of a system.

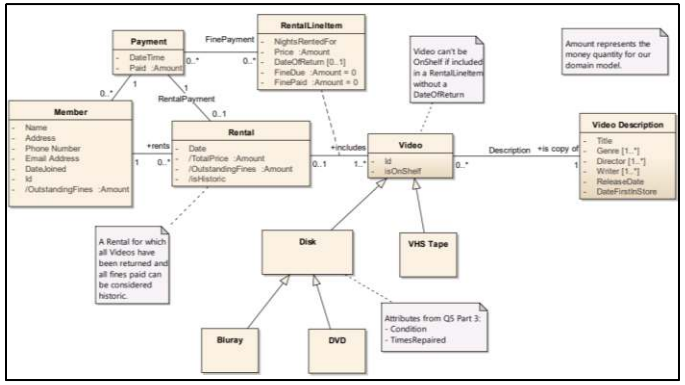
**Types of Actors**
**1. Primary**: Uses services of the SuD
**2. Supporting**: provides a service to the SuD
**3. Offstage**: Has an interest in behaviour of the use case, but is not primary or supporting

**Tests for "Useful" Use Cases**
1. *Boss Test*: would your boss be happy if you told them the use case you've been working on all day?
2. *Elementary Business Process Test*: should be a task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state
3. *Size Test*: Very seldom a single action/step

## INTERACTION DIAGRAMS

| Type | Strengths | Weaknesses |
|---|---|---|
| sequence | Clearly shows time ordering of messages<br><br>Can more easily convey the detail of message protocols between objects | Linear layout of instances can obscure relationships<br><br>Linear layout consumes horizontal space |
| communications | More layout options<br><br>Clearly shows relationships between object instances<br><br>Can combine scenarios to provide a more complete picture | More difficult to see message sequencing<br><br>Fewer notation options for expressing message patterns |


sd Return damaged disk
:Assistant
:System
Inspect and get non-damaged disks back on shelf quickly. Deal with repairs/reorders when time permits.
DiskScratches= InspectedForStratches(VideoId) :{None,Minor,Major}
isDamaged= ReturnDisk(VideoId, DiskScratches) :Boolean
AddToDamagedPile(VideoId)
IsDead= CheckIfDead(VideoId) :Boolean
AttemptingRepair(VideoId)
DiskScratches= InspectedForScratches(VideoId) :{Minor,Major}
IsDead= RepairResult(VideoId, DiskScratches) :Boolean
IsPopular= AssessIfPopular(VideoId) :Boolean
Reorder(VideoId)
ThrowAway(VideoId)
DiskScratches == Major
IsDamaged == true
IsDead == false
DiskScratches == Major
IsDead == true
IsPopular == true



## ITERATIVE, EVOLUTIONARY, AGILE

- Involves early programming and testing of a partial system, in repeating cycles (build-feedback-adapt).
- Also assumes development starts before all the requirements are defined in detail.
- Feedback is used to clarify and improve the evolving specifications.

In an iterative approach:
- Development is organised into a series of short, fixed-length iterations
- The outcome of each iteration is a tested, integrated and executable *partial* system, which is a production grade subset of the final system.
- Each iteration includes its own requirements analysis, design, implementation and testing.

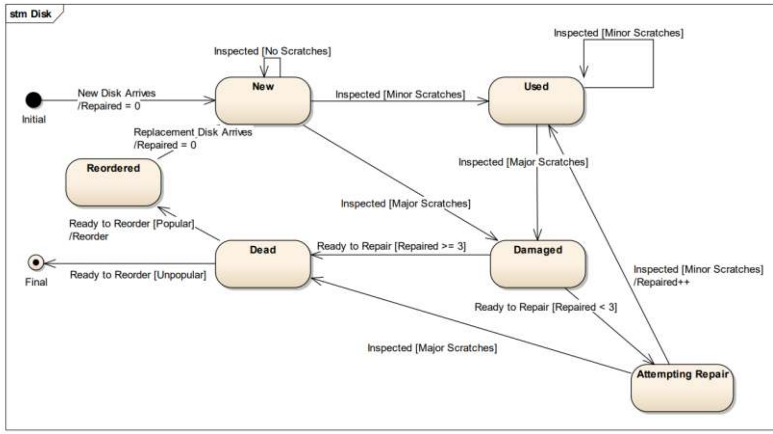
**Iterative Advantages:** (1) early visible progress (2) early feedback, adaptation & user engagement (3) managed complexity

**Waterfall Development**: progress flows largely in a linear fashion through the phases of conception, initiation, analysis, design, implementation, testing, maintenance etc.

## STATE MACHINES

State Machine models can be very useful especially when trying to deliver secure and safe software. They provide a perspective on system design that helps to identify the necessary guards to put in place when implementing a system. They help us to identify what precautions need to be put in place to prevent invalid state transitions and ensure system correctness and safety.

When object is in *State A*:
  if *trigger* event occurs and *guard* is true
  then
    perform the behaviour *action* and
    transition object to *State B*.


stm Disk
Initial
New Disk Arrives /Repaired = 0
New
Inspected [No Scratches]
Inspected [Minor Scratches]
Used
Inspected [Minor Scratches]
Replacement Disk Arrives /Repaired = 0
Reordered
Inspected [Major Scratches]
Inspected [Major Scratches]
Ready to Reorder [Popular] /Reorder
Dead
Ready to Repair [Repaired >= 3]
Damaged
Final
Ready to Reorder [Unpopular]
Inspected [Minor Scratches] /Repaired++
Ready to Repair [Repaired < 3]
Inspected [Major Scratches]
Attempting Repair

## ARCHITECTURE

**Software architecture**: The set of *significant decisions* about the organisation of a software system. The selection of *structural elements* (e.g. classes) and *interfaces* (e.g. public methods) by which the system is composed.
**Logical Architecture**: The large-scale organisation of the software classes into packages, subsystems and layers. *Logical* refers to not being concerned with networking, physical computers, or operating system processes.
**Layer**: A coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system. Coupling and cohesion are just as important at a larger scale/higher level.
**Distributed Architectures**: Components typically are hosted on different platforms and communicate through a network.

**Design Guideline:**
- Organise large scale logical structure into distinct cohesive layers
- High layers – application specific
- Low layers – general services
- High level layers can depend on low level layers but not vice versa; low to high level coupling avoided

E.g. UI, DOMAIN, TECHNICAL SERVICES, FOUNDATION

**Problems Addressed by Layering**: (1) Changes rippling through system due to coupling. (2) Intertwining application logic and UI, reducing reuse & restricting distribution options. (3) High coupling across areas of concern, impacting division of development work

**Model-View Separation Principle**
1. Don't couple non-UI objects directly to UI objects
2. Don't put application logic in UI object methods

Relaxation of this principle via the *Observer pattern*.

## ARCHITECTURAL ANALYSIS

Architectural Analysis: *Identifying and resolving a system's non-functional requirements in the context of its functional requirements*. It reduces the risk of missing a critical factor in the system design and helps focus effort.

Includes identifying & analysing:
(1) Architecturally significant requirements, (2) variation points (3) probable evolution points (alternatives in the future).

**Architecturally Significant Non-Functional Requirements**:
(1) Usability (2) Reliability (3) Performance (4) Supportability

**Steps in Architectural Analysis**
(1) Identify/analyse *architectural factors*. Some identified during *inception* but not investigated in more detail.
(2) Analyse alternatives and create solutions, called *architectural decisions* (e.g. remove requirement; custom solution; stop project; hire expert).

**Priorities**: (a) Inflexible constraints - *e.g. security, safety, legal*, (b) Business goals, (c) Other goals.

**Architectural Factor Table**:
Understand the influence of architectural factors including their priorities and variability.
1.Factor
2.Measures and quality scenarios
3.Variability
4.Impact of factor (and variability) on stakeholders and architecture
5.Priority for success
6.Difficulty or risk

**Technical Memo**: Helpful to understand the motivations behind the design, such as why a particular approach was chosen and others rejected.
1.Issue
2.Solution Summary
3.Factors
4.Solution
5.Motivation
6.Unresolved Issues
7.Alternatives Considered

**Summary**
(1) Concerns related to non-functional requirements with awareness of business context, addressing functional requirements and their variability.

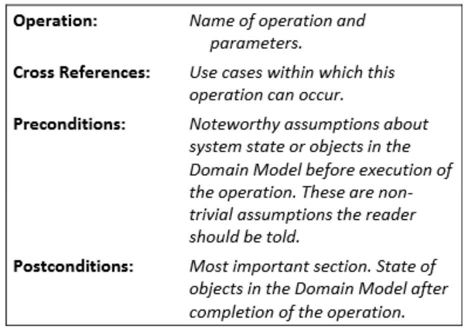
(2) Concerns involve system-level, large-scale, broad problems, with resolution involving large-scale fundamental design decisions.

(3) Must address interdependencies and trade-offs, which involves generation/evaluation of alternatives.

## OPERATIONS CONTRACTS

Use a pre- and post-condition form to describe detailed changes to objects in a domain model, as the result of a system operation.

**Advantages:** discover the need to record new conceptual classes, attributes, or associations in the domain model.

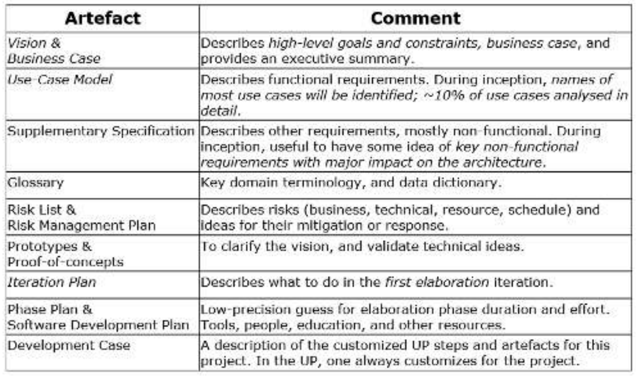


| | |
|---|---|
| **Operation:** | *Name of operation and parameters.* |
| **Cross References:** | *Use cases within which this operation can occur.* |
| **Preconditions:** | *Noteworthy assumptions about system state or objects in the Domain Model before execution of the operation. These are non-trivial assumptions the reader should be told.* |
| **Postconditions:** | *Most important section. State of objects in the Domain Model after completion of the operation.* |

## INCEPTION PHASE

Initial short project phase answering questions like:
(1) Vision and business case for this project? (2) Feasible? (3) Buy and/or build? (4) Rough unreliable range of cost? (5) Proceed/Stop?
*i.e. do stakeholders have a basic agreement on the vision of the project, and is it worth investing in serious investigation?*

**Outcome:**
(1) Common vision and basic scope for the project (2) Go or no go decision (3) Analysis of ~10% of use cases (4) Analysis of critical non-functional requirements (5) Preparation of the development environment



| Artefact | Comment |
|---|---|
| *Vision & Business Case* | Describes *high-level goals and constraints, business case*, and provides an executive summary. |
| *Use-Case Model* | Describes functional requirements. During inception, *names of most use cases will be identified; ~10% of use cases analysed in detail.* |
| Supplementary Specification | Describes other requirements, mostly non-functional. During inception, useful to have some idea of *key non-functional requirements with major impact on the architecture.* |
| Glossary | Key domain terminology, and data dictionary. |
| Risk List & Risk Management Plan | Describes risks (business, technical, resource, schedule) and ideas for their mitigation or response. |
| Prototypes & Proof-of-concepts | To clarify the vision, and validate technical ideas. |
| *Iteration Plan* | Describes what to do in the *first elaboration* iteration. |
| Phase Plan & Software Development Plan | Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources. |
| Development Case | A description of the customized UP steps and artefacts for this project. In the UP, one always customizes for the project. |

## SOFTWARE PATTERNS

**Pattern:** a recurring successful application of expertise in a particular domain that can be applied independently of particular technology or tools.

*Advantages:* (1) capture expertise, make it accessible (2) facilitate communication by providing common language
(3) make it easier to reuse successful applications of expertise

**Responsibility**: A *contract* or *obligation* of a *classifier*. A classifier is a class/interface or other component of the software system. We want to make components *responsible* for some part of the system.

**Responsibilities** and **Methods** are related but not the same: Responsibility → abstract concept; Methods → concrete ways of achieving responsibilities

**Responsibility-Driven Design (RDD)** sees an OO design as a *community of collaborating, responsible objects.* Involves assigning responsibilities to classes based on proven principles.

**Data-Driven Design:** cover a broad family of techniques including reading codes, values, class file paths, class names, and so forth, from an external source in order to change the behaviour of, or "parameterize" a system in some way at run-time.

## GRASP PATTERNS

**Controller**: Relates to which object receives the first message from an **external system** and coordinates or controls a **system operation** (major input event, appears on SSD). Assign responsibility to a class representing one of: (1) the **overall system**, a **"root" object**, a **device** the system is running in, or a **major subsystem**. E.g. a *facade controller* → representation of system as a whole with routing point where all messages go through.
(2) A use case scenario or session controller that deals with the event in complex systems. The controller lives just under the UI layer in the domain layer. The UI layer needs to know where to route the message to.

**Contraindications**: Can sometimes get a **bloated controller** with too many responsibilities and low cohesion. The controller becomes too complicated with too much logic and duplicated information. Solutions: (a) May need to add more controllers based on use case. (b) Delegate to responsibility.

**Creator**: Deals with problem of which class is responsible for creating a new instance of a class. The action of creating another class is an assignment of responsibility.
We want to ensure *low coupling*, *increased clarity*, *encapsulation* and *reusability*.

**B** should be responsible for creating **A** if: (a) **B** contains or compositely aggregates **A** (most important), (b) **B** records **A**, or (c) **B** has the initialising data for **A** (B is an expert).

**Contraindications**: May decide against the creator pattern in the case of: Creation of significant complexity (e.g. When recycling instances for performance (taking objects from a pool; before creating new ones; Only retrieving specific instances based on property). In these cases we can delegate to a helper class in the form of a
*Concrete Factory* or *Abstract Factory*.

**High Cohesion**
**Problem:** how to keep objects focused, understandable & manageable, and as a side effect, support low coupling. *Cohesion* is a measure of how strongly (functionally) related & focused the responsibilities of an element are.
**Contraindications**: (a) Lower cohesion sometimes required to meet non-functional requirements such as performance (e.g. reduce processing overheads).

**Indirection**
**Problem:** Where to assign responsibility to avoid direct coupling between two or more software elements?
**Solution:** De-coupling components so that low coupling is supported and reuse potential remains high by assigning the responsibility to an **intermediate object**. The intermediary creates an *indirection* between the other components.

**Information Expert**
**Problem:** What is a general principle of assigning responsibilities to objects?
**Solution:** Assign responsibility to the class that has the information necessary to fulfil the responsibility.
**Contraindications**: Don't use expert if the suggested solution results in poor coupling or cohesion.

**Low Coupling**: Coupling is a measure of how strongly one element is **connected to, has knowledge of, or relies on** others. It relates to the connections between components in a software system. More connections → harder to maintain. Low coupling suggests assigning responsibility so that the design supports **low dependency, low change impact and increased reuse**. The lower the dependency, the less changes propagate.
**Contraindications:** (a) High coupling to stable elements isn't an issue (b) Coupling is actually essential, so we are just choosing to keep it low where;
(c) we can without compromising other design aspects.

**Polymorphism**: If we purely use if-else's, we get a proliferation of them. Need a better way of handling alternatives based on type. Allows us to handle alternatives based on type by giving a single interface to entities of different types. When related alternatives differ by type (class), we use operations with the same interface to assign responsibilities for the behaviour to the types for which the behaviour varies.
Polymorphism helps us to create *pluggable software components*.
*Polymorphic* → "giving a single interface to entities of different types"
*Adapters* → objects responsible for handling varying interfaces.

**Protected Variations**: A way of designing objects/systems so that variations/instability in these elements do not have an undesirable impact on other elements. Identify known or predicted variation/instability and assign responsibilities to create a stable interface around them. An interface refers to **a means of access**, it's not just a programming language interface. Points of change include:
• **variation points**: variations in existing system - *e.g. multiple tax calculators*
• **evolution points**: speculative variations that may arise in the future

**Open-Closed Principle (OCP)** - *strongly related to protected variation*. Modules should be both open (for extension; adaptable) and closed (to modification that affects clients) e.g. define a class and only use access methods → can change field definitions without impacting clients.
**Contraindications**: Cost of speculative "future-proofing" at evolution points can outweigh the benefits. Can be cheaper/easier to rework a simple "brittle" design.

**Pure Fabrication**
**Problem:** Which object should have responsibility when solutions offered (say, by Expert) violate High Cohesion and Low Coupling?
**Solution:** Can **fabricate an object** not in the domain to give a better design. Required when a low representation gap/solution offered by expert causes high coupling/low cohesion. Only to be used when

other techniques don't work. E.g. Creating a PersistentStore for saving DB related transactions like a Sale.
**Contraindications**: (a) Sometimes driven by behavioural decomposition into functions, resulting in functions being group into objects. Don't just create an object when looking for a place to put a quick set of functions.

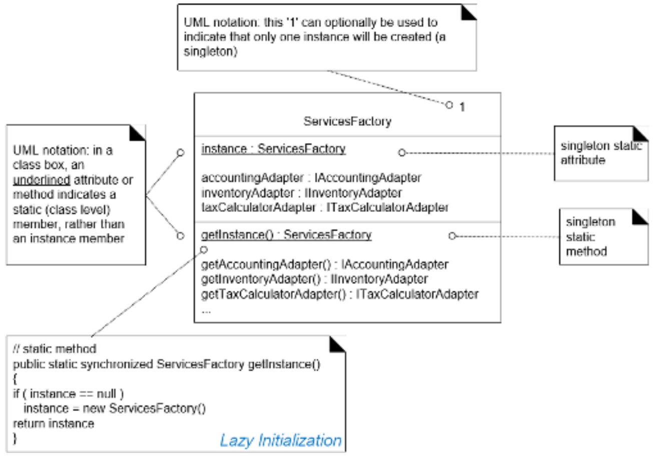
## GANG OF FOUR PATTERNS

**Creational Patterns**: Patterns to abstract the object instantiation process.
**Structural Patterns**: Patterns to combine classes and objects to form larger structures.
**Behavioural Patterns**: Patterns to manage communication between objects.

### Adapter
***Problem***: How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
***Solution***: Convert the original interface of a component into another interface, through an intermediate adapter object.
***Related patterns***
***GoF** - **Facade** - provides a single coherent interface to a subsystem with a single object, as opposed to an adapter which requires polymorphism to deal with varying external systems (multiple potential/actual systems we want to deal with. Hiding detail of multiple subsystems)*
***GRASP - Protected Variations** (PV) - in respect to changing external interfaces or 3rd party packages.* ***Indirection*** - an object that provides indirection through an interface to achieve low coupling and support PV**.** ***Polymorphism***: achieve PV at a variation point

### Concrete Factory
***Problem***: Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities etc.?
***Solution***: Create a Pure Fabrication object called a Factory that handles the creation.
*Advantages*: Separate responsibility of complex creation into cohesive helper objects.
Hide potentially complex creation logic. Allow introduction of performance-enhancing memory management strategies, such as *object caching* or *recycling* (reallocation of objects). Save it into a properties file and load. Now the only thing we need to change to change the adapter the system uses is edit the property file.

### Singleton
***Problem***: How do we create exactly one instance of a class with objects that need a global and single point of access?
***Solution***: Define a static method of the class that returns the singleton. Needs to be synchronized to lock out multiple processes from trying to create at the same time.  Singleton class is globally visible and accessible.

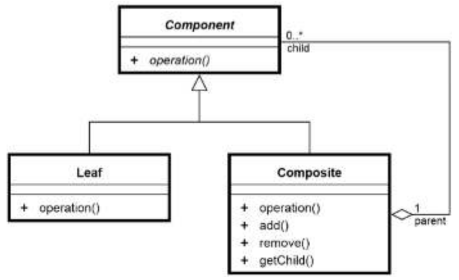

UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)
1
ServicesFactory
instance : ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getInstance() : ServicesFactory
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member
singleton static attribute
singleton static method
Lazy Initialization

```
// static method
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
   instance = new ServicesFactory()
return instance
}
```

Don't want to make all Singleton methods static, as static methods are not polymorphic (virtual), so can't override. Most remote communication mechanisms don't support remote-enabling of static methods either.

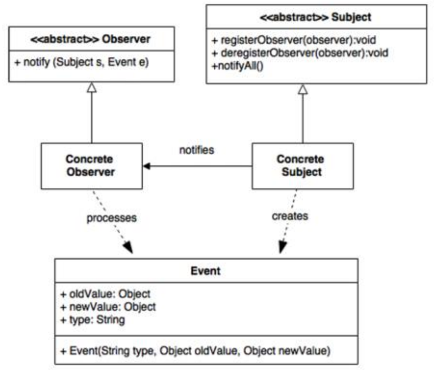
### Strategy
***Problem***: How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies? We are looking to vary some behavioural element.
***Solution***: Define each algorithm/policy/strategy in a separate class with a common interface. This looks like the adapter pattern, but with a focus on behaviour. Each class implements different behaviour, it's not just an interface for translating. With the strategy pattern, use a 'context' object e.g. pass Sale to PercentDiscountPricingStrategy.

### Composite
***Problem***: How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?
***Solution***: Define classes for the composite and atomic objects so that they implement the same interface.



### Façade
***Problem*:** We require a common, unified interface to a disparate set of implementations or interfaces. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change. What should we do?
e.g. The subsystem might be a complex rule engine
***Solution*:** Define a single point of contact to the subsystem - a facade object that wraps the subsystem. This facade objects presents a single unified interface and is responsible for collaborating with the subsystem components.

### Observer
***Problem***: Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way. The publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. How?
***Solution***: Define a *subscriber* or *listener* interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.


<<abstract>> Observer
+ notify (Subject s, Event e)
<<abstract>> Subject
+ registerObserver(observer):void
+ deregisterObserver(observer):void
+notifyAll()
Concrete Observer
notifies
Concrete Subject
processes
creates
Event
+ oldValue: Object
+ newValue: Object
+ type: String
+ Event(String type, Object oldValue, Object newValue)