CISSA Revision Workshop
Software Modelling & Design

Sem 1 2022 - Kyla Canares, Charlie Ding

https://www.facebook.com/cissa.unimelb/
https://cissa.org.au/

Welcome :)

Brought to you by CISSA!

Largest Tech Club at UoM

Sponsored by companies such as Google, Microsoft, Canva, Optiver, Amazon

Presenters
Kyla Canares- https://www.linkedin.com/in/kyla-canares-4773a7187/
Charlie Ding- https://www.linkedin.com/in/charlie-ding21/

Sign-up!

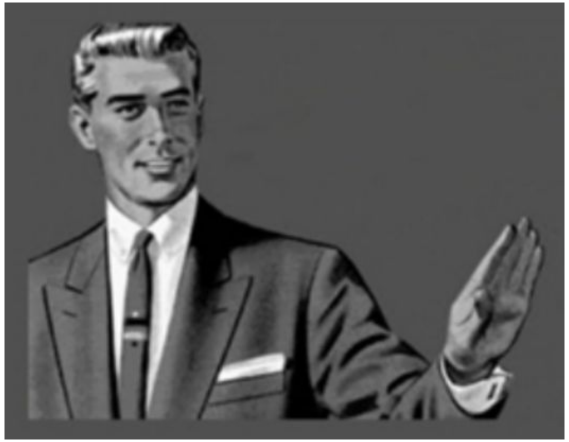
Recording and slides will be posted at:
https://www.facebook.com/cissa.unimelb/videos
Join the CISSA Discord!
https://discord.gg/enY8Tt74

Agenda!

- 4:30 - 4:50 - Diagram breakdown + modelling
- 4:50 - 5:20 - GRASP
- 5:20 - 5:50 - Gang of Four patterns
- 5:50 - 6:20 - Exam questions
- 6:20 - 6:30 - Q&A

Disclaimer

- This is not an exhaustive guide
- The single source of truth is the subject material
- We are volunteers, not being paid, not representing the subject officially

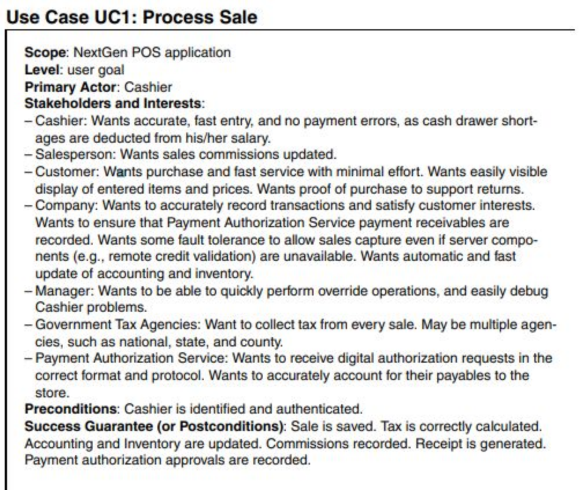


Use Cases

Use Cases: textual

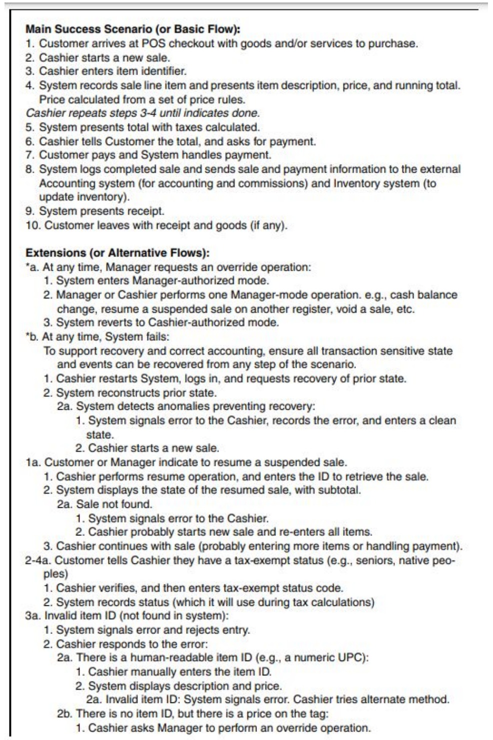
Brief	Casual	Fully dressed
Paragraph of text	Has subheadings: - Main scenario, alternative scenario	Has more subheadings: - E.g. Preconditions, primary actor etc Has numbered steps

Fully dressed example



Use Case UC1: Process Sale

Scope: NextGen POS application
Level: user goal
Primary Actor: Cashier
Stakeholders and Interests:
– Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
– Salesperson: Wants sales commissions updated.
– Customer: Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.
– Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
– Manager: Wants to be able to quickly perform override operations, and easily debug Cashier problems.
– Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
– Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
Preconditions: Cashier is identified and authenticated.
Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.



Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

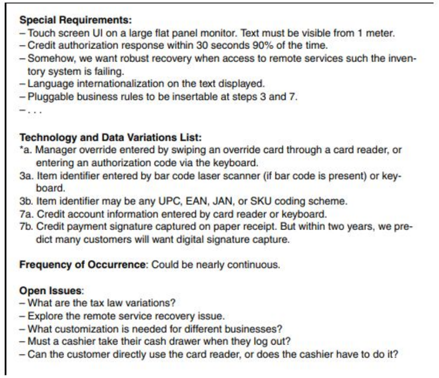
*a. At any time, Manager requests an override operation:
- 1. System enters Manager-authorized mode.
- 2. Manager or Cashier performs one Manager-mode operation. e.g., cash balance change, resume a suspended sale on another register, void a sale, etc.
- 3. System reverts to Cashier-authorized mode.

*b. At any time, System fails:
- To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.
- 1. Cashier restarts System, logs in, and requests recovery of prior state.
- 2. System reconstructs prior state.
  - 2a. System detects anomalies preventing recovery:
    - 1. System signals error to the Cashier, records the error, and enters a clean state.
    - 2. Cashier starts a new sale.

1a. Customer or Manager indicate to resume a suspended sale.
- 1. Cashier performs resume operation, and enters the ID to retrieve the sale.
- 2. System displays the state of the resumed sale, with subtotal.
  - 2a. Sale not found.
    - 1. System signals error to the Cashier.
    - 2. Cashier probably starts new sale and re-enters all items.
- 3. Cashier continues with sale (probably entering more items or handling payment).

2-4a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples)
- 1. Cashier verifies, and then enters tax-exempt status code.
- 2. System records status (which it will use during tax calculations)

3a. Invalid item ID (not found in system):
- 1. System signals error and rejects entry.
- 2. Cashier responds to the error:
  - 2a. There is a human-readable item ID (e.g., a numeric UPC):
    - 1. Cashier manually enters the item ID.
    - 2. System displays description and price.
      - 2a. Invalid item ID: System signals error. Cashier tries alternate method.
  - 2b. There is no item ID, but there is a price on the tag:
    - 1. Cashier asks Manager to perform an override operation.



Special Requirements:
– Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
– Credit authorization response within 30 seconds 90% of the time.
– Somehow, we want robust recovery when access to remote services such the inventory system is failing.
– Language internationalization on the text displayed.
– Pluggable business rules to be insertable at steps 3 and 7.
– . . .

Technology and Data Variations List:
*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.
3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
7a. Credit account information entered by card reader or keyboard.
7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:
– What are the tax law variations?
– Explore the remote service recovery issue.
– What customization is needed for different businesses?
– Must a cashier take their cash drawer when they log out?
– Can the customer directly use the card reader, or does the cashier have to do it?

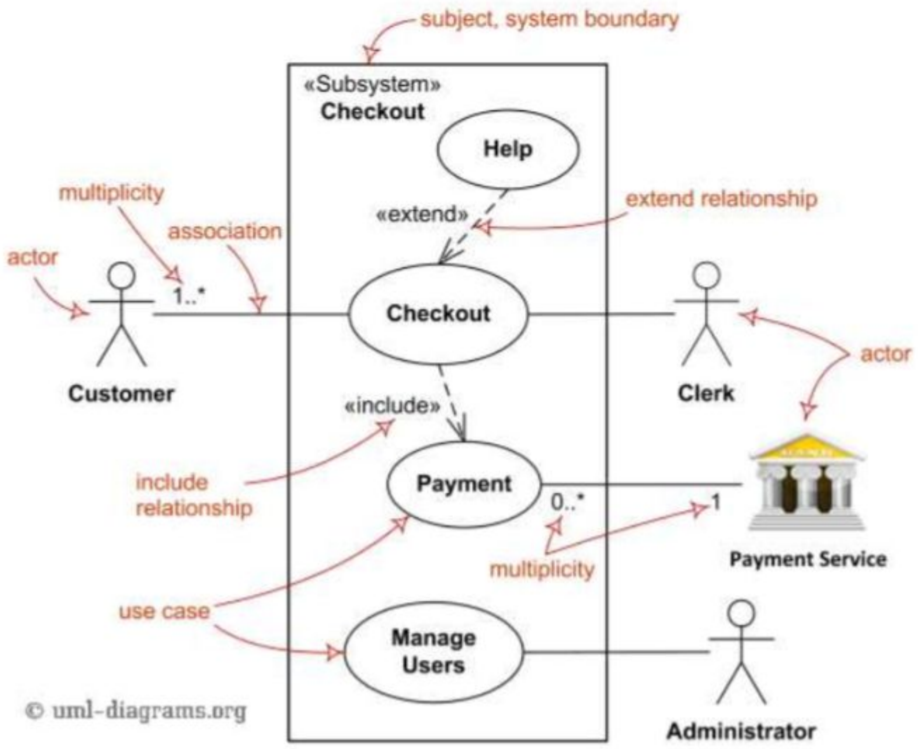
https://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf

Use case: tests

- **Boss Test:** If I told my boss I was doing x would they be happy? If not, not a use case
- **Size Test:** If the use case has too few steps it potentially should be part of a use case rather than by itself.
- **Elementary Business Process Test (EBP):** 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

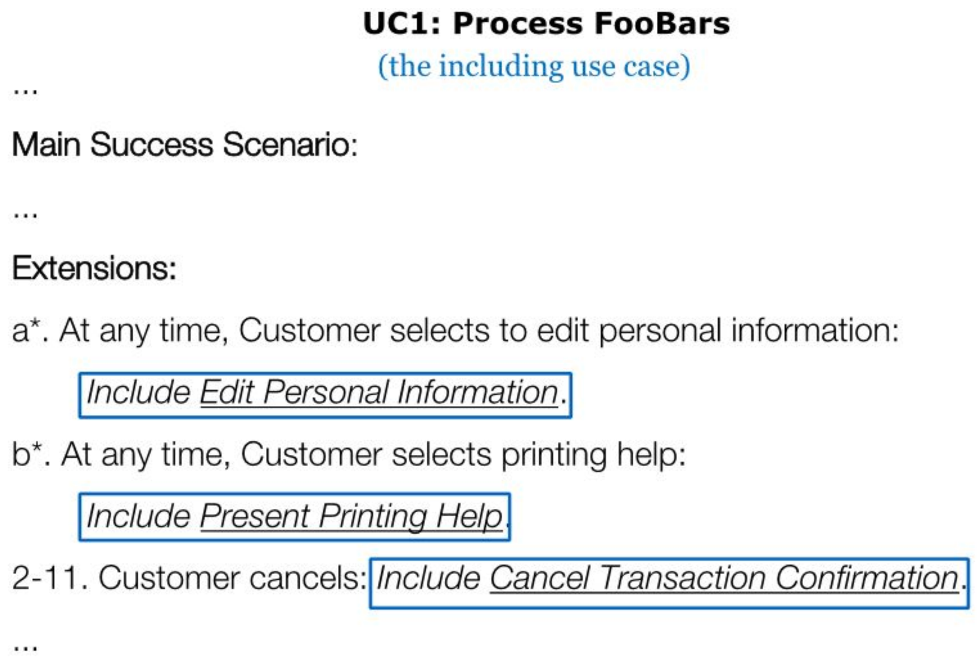
Use Cases: diagram

- What's the difference between the diagram and the text?
  - Use case diagrams include **all** use cases in a subsystem and show their relationships
  - The text contains only **one** use case
- What's the difference between extend and include?
  - Include is for reducing repetition in the text
  - Base case should be complete without extension



Include vs Extend

The ‘Include’ in this textual use case would be represented as ‘Include’ in the use case diagram



UC1: Process FooBars

(the including use case)

...

Main Success Scenario:

...

Extensions:

a*. At any time, Customer selects to edit personal information:

Include Edit Personal Information.

b*. At any time, Customer selects printing help:

Include Present Printing Help.

2-11. Customer cancels: *Include Cancel Transaction Confirmation.*

...

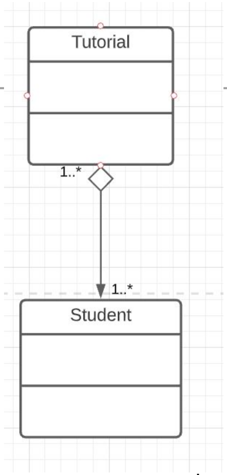
Static diagrams

Domain vs Design Class Diagram

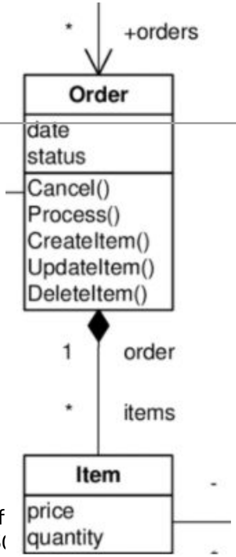
Domain	Design
● No software objects ● Can have inheritance (clear arrows) ● Should include the "system" itself. ○ E.g. in assignment 1, you should have SLOP as an entity in the diagram ● Should show all "semantic" relationships.	● Has extra coding detail ○ E.g. visibility modifiers ● Has classes that are codeable ● Get rid of semantic relationships and introduce classes for better coupling and cohesion ○ Recall house marketplace example from the workshops

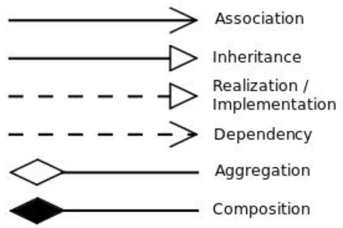
Design diagrams: aggregation vs composition

Aggregation	Composition
● assembles together children to form a more complex object, but the children can exist without the parent	● assembles together children and the children would not exist at all without the parent



https://www.researchgate.net/figure/A-UML-class-diagram-of
ent-system-The-class-diagram-contains-the-main_fig1_22125





https://federicohaag.medium.com/uml-unified-modelin
g-language-5a2a0c2fb973

Dynamic diagrams

System vs Design

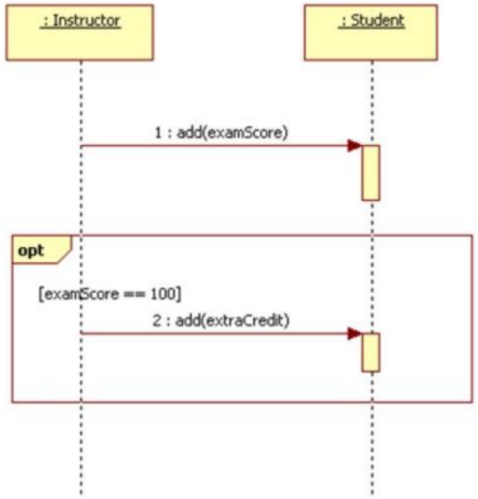
System Sequence Diagram	Design Sequence Diagram
- one diagram should be one use case. - visualises black boxes, no code methods - has interactions between actor + system(s)	- Has software concepts - E.g. methods, how the flow of logic works in code, frames, messages - entry point, exit point, inter-class/subsystem interaction

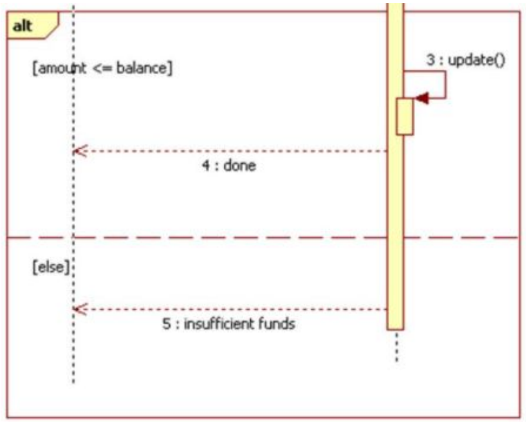
Easier to make a design sequence diagram when you have a static design class diagram

Design Seq Diagram Fragments: opt and alt

Optional → **opt**. One-way conditional

Alternative → **alt**. Multi-way conditional





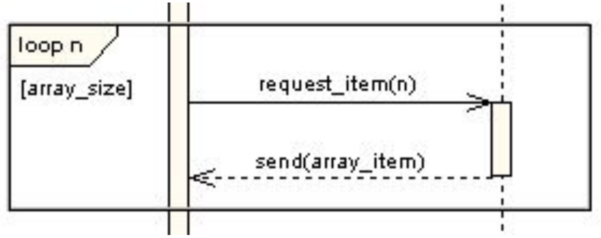
Don’t leave empty boxes!!!

https://sparxsystems.com/images/screenshots/uml2_tutorial/seq07.GIF

http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/uml/interaction/SequenceDiagrams.htm

Design Seq Diagram Fragments: loop and ref

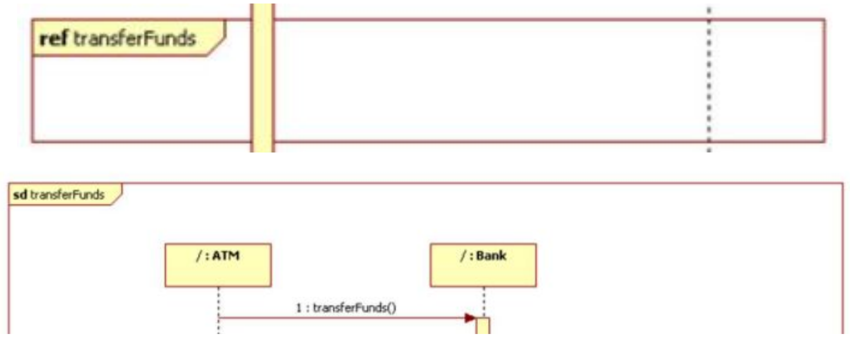
Loop → **loop or while**. Repeat until guard fails



https://sparxsystems.com/images/screenshots/uml2_tutorial/seq07.GIF

Reference frames → **ref**. Refer to another diagram

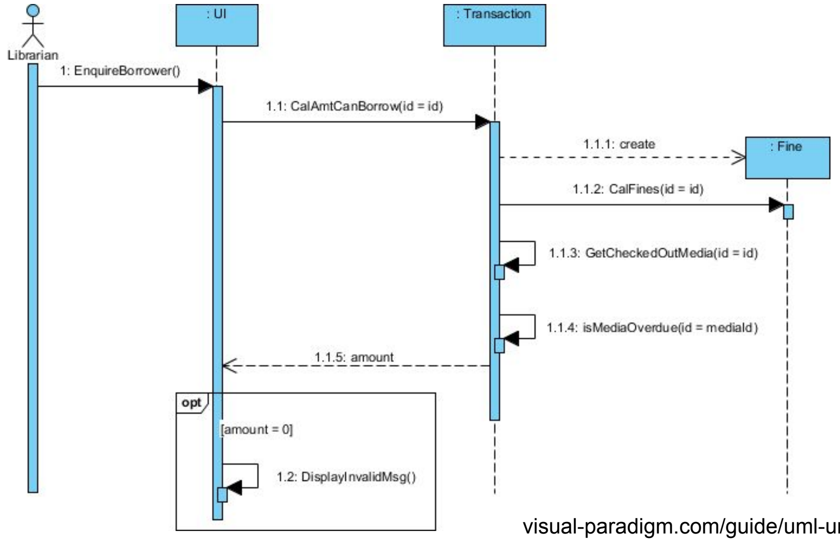
- Ensure referenced diagram has a "**sd**" box around it

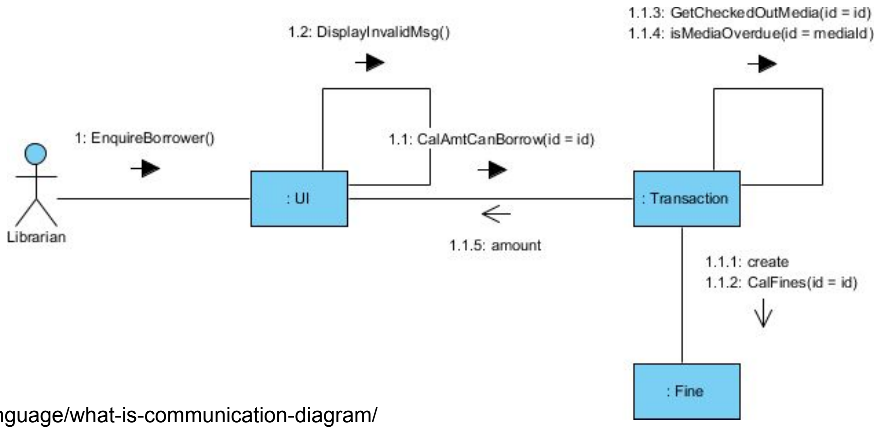


http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/uml/interaction/SequenceDiagrams.htm

Communication diagrams

- Basically a non-linear design sequence diagram
- Shown below is an example of moving from design sequence diagram to communication diagram
- Numbering of steps
- Pro -- Better shows relationships between object instances
- Con -- Doesn't easily show code behaviour → loop, alternate, optional



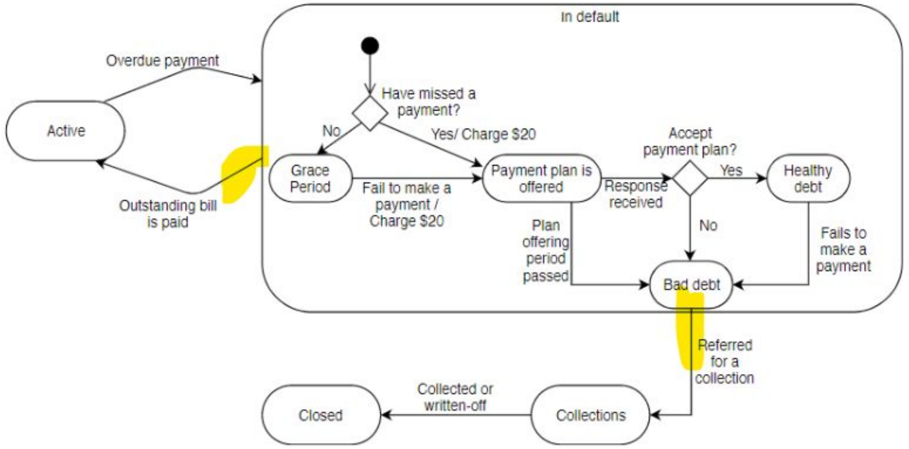


visual-paradigm.com/guide/uml-unified-modeling-language/what-is-communication-diagram/

State machine diagrams

State Machine Diagrams

- Each **circle is a state**, arrows are events
- Check out the reference guide on the lms
- Note the two ways you can exit a nested state
- Going to code
  - States usually become Java enums
  - See workshop 6 video on LMS



Architecture

Architecturally Significant Requirements

Functional

- **WHAT the system does (or must not do)**
- Affects inputs/outputs
- Requires interfacing with external system
- Basic system functionality
- User requirements

- Examples:
  - Auditing
  - Mail service
  - Security
  - Reporting

Non-functional

- **The quality of the system**
- Does not affect basic functionality
- Product properties
- User expectations

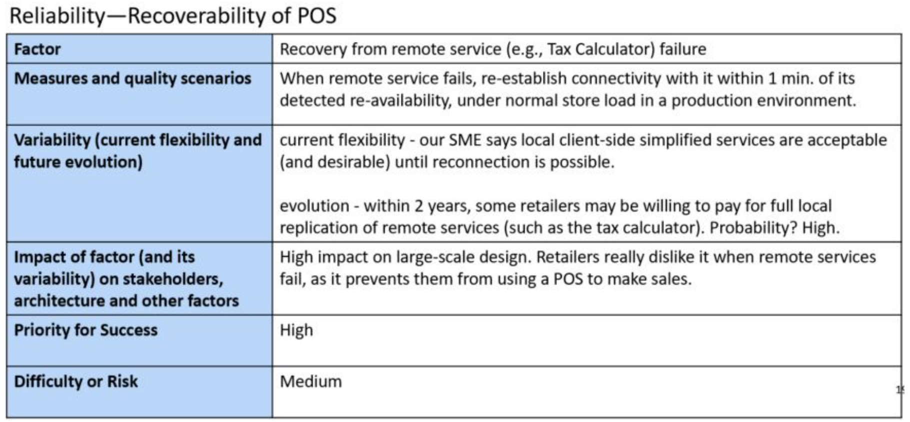
- Examples:
  - Usability
  - Reliability
  - Performance
  - Supportability

https://qracorp.com/functional-vs-non-functional-requirements/

Technical Memos & Architectural Factor Tables

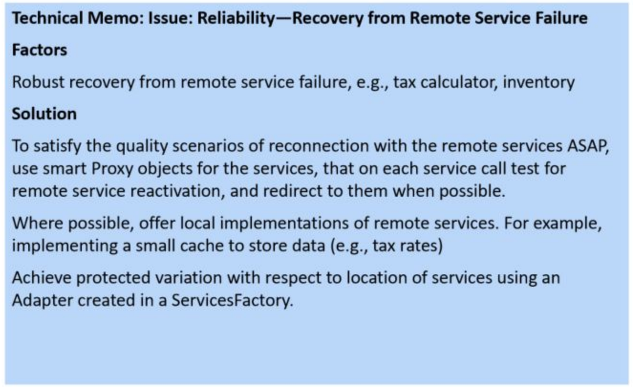
- **Architectural Factor Table**
  - records influence of architectural factors
  - E.g. our customer base is the elderly, how do we make sure that our software is useable?

- **Technical Memo**
  - documentation that records alternate solutions, and the decision making process



Reliability—Recoverability of POS

Factor	Recovery from remote service (e.g., Tax Calculator) failure
Measures and quality scenarios	When remote service fails, re-establish connectivity with it within 1 min. of its detected re-availability, under normal store load in a production environment.
Variability (current flexibility and future evolution)	current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible. evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High.
Impact of factor (and its variability) on stakeholders, architecture and other factors	High impact on large-scale design. Retailers really dislike it when remote services fail, as it prevents them from using a POS to make sales.
Priority for Success	High
Difficulty or Risk	Medium



Technical Memo: Issue: Reliability—Recovery from Remote Service Failure

Factors

Robust recovery from remote service failure, e.g., tax calculator, inventory

Solution

To satisfy the quality scenarios of reconnection with the remote services ASAP, use smart Proxy objects for the services, that on each service call test for remote service reactivation, and redirect to them when possible.

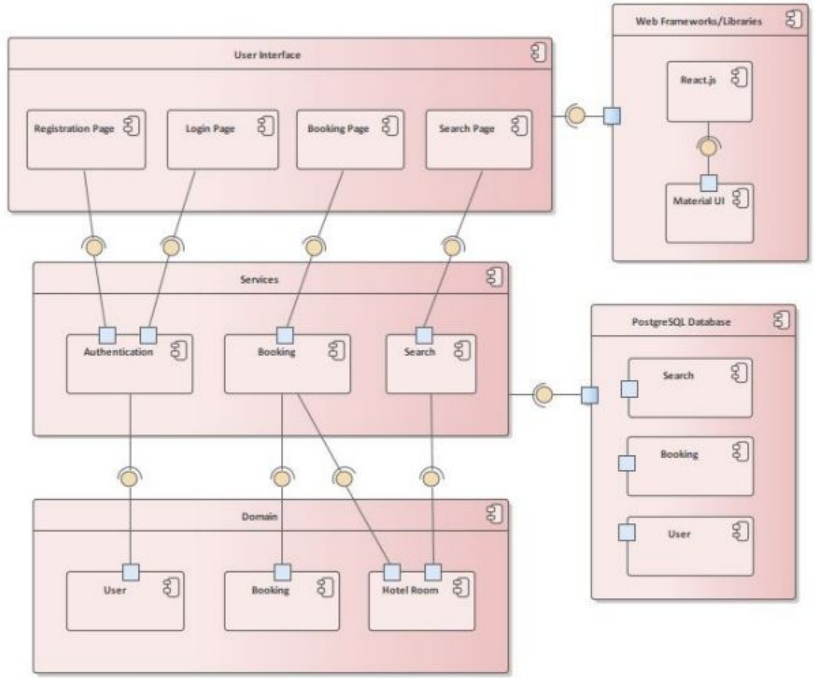
Where possible, offer local implementations of remote services. For example, implementing a small cache to store data (e.g., tax rates)

Achieve protected variation with respect to location of services using an Adapter created in a ServicesFactory.

Both documents are especially concerned with non-functional requirements

Component diagram

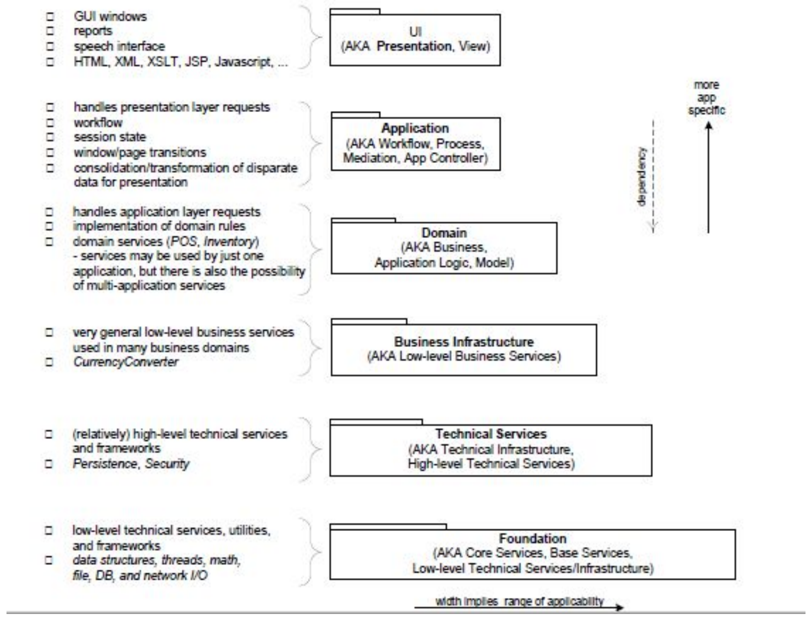
- Shows how the architecture components interacts with each other
  - components can be a class, external resource, service, or subsystem
- See notation reference on lms



https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/

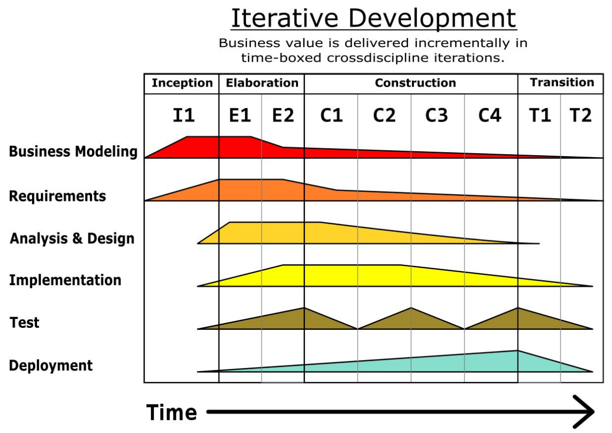
Layered architecture

- One way of organising architecture components
- Different languages or technologies may be used for different layers
- Often referred to as an application stack


GUI windows
reports
speech interface
HTML, XML, XSLT, JSP, Javascript, ...
UI (AKA Presentation, View)
handles presentation layer requests
workflow
session state
window/page transitions
consolidation/transformation of disparate data for presentation
Application (AKA Workflow, Process, Mediation, App Controller)
handles application layer requests
implementation of domain rules
domain services (POS, Inventory) - services may be used by just one application, but there is also the possibility of multi-application services
Domain (AKA Business, Application Logic, Model)
very general low-level business services used in many business domains
CurrencyConverter
Business Infrastructure (AKA Low-level Business Services)
(relatively) high-level technical services and frameworks
Persistence, Security
Technical Services (AKA Technical Infrastructure, High-level Technical Services)
low-level technical services, utilities, and frameworks
data structures, threads, math, file, DB, and network I/O
Foundation (AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)
more app specific
dependency
width implies range of applicability

Unified Process (UP)

- **Inception** - period of ideation and rough design for software product. Is it possible?
- **Elaboration** - product is more heavily modelled out and designed, requirements captured in user stories
- **Construction** - product is built, requirements can be flexible and change as development progresses through iterations (sprints)
- **Transition** - handover product to client or new maintenance team



https://en.wikipedia.org/wiki/Unified_Process

GRASP

Creator

Use Cases:

Assign class B responsibility to create instances of class A if one of these is true (the more the better):

– B “contains” or compositely aggregates A

– B records A.

– B closely uses A

– B has the initializing data for A.

Contraindications:

Complex creation cases such as:

- Creating objects conditionally (which type of sprite in a game)
- Reusing objects for performance gain

Should be abstracted to a concrete or abstract factory

Information Expert

Use Cases:

Assign the responsibility to an object if the object already has the information to fulfil the role

Makes code more understandable, maintainable and extensible

Contraindications:

If the responsibility requires complex dependencies:
(ie. manipulating database values)

Then the complex responsibility that includes dependencies should be done by helper classes.

Controller

Use Cases:

Assign responsibility to a class representing one of

- The overall system or root object/major subsystem
- A use case scenario that deals with a specific event
- Use case or session controller

Contraindications:

Controllers become bloated as system becomes more complex and more use cases are added.

Delegate responsibility to other classes and use facade to manage.

Polymorphism

Use Cases:

When related entities differ by type, use polymorphism to capture the shared behaviour while representing the alternatives.

Contraindications:

Speculative polymorphic design can introduce unnecessary complexity into a system that does not require it.

Ask if variation between entities is significant enough to warrant polymorphism.

Indirection

Use Cases:

When coupling is too high, assign responsibility to an intermediate object to mediate so that the original classes are not directly coupled.

The intermediary creates and indirection between the other components

Contraindications:

Unnecessary indirection can create many software artifacts and introduce unneeded complexity

When using indirection be careful to adequately signpost it, so that readers can see that it is occurring - otherwise code can be difficult to work with.

GRASP - Pure Fabrication

Use Cases:

If functionality needs to be introduced that is not represented in the domain model, then assign a cohesive set of responsibilities to an artificial or convenience class (helper) that does not represent a domain concept.

Note: indirection and pure fabrication can both create new classes but indirection is more concerned with coupling while PF is concerned with cohesion

Contraindications:

Overuse of pure fabrication can end up with one-class-one-responsibility objects

Coupling can become high with too many fabricated objects.

Low Coupling

Use Cases:

Assign responsibilities to classes with the aim of keeping coupling to other classes low.

Often later step when evaluating otherwise equal alternatives but should be kept in mind during all design decisions.

The less coupled code is the easier it is to maintain. This is because classes are less dependant and the impact of change is reduced.

Contraindications:

When elements are very stable and almost never change, high coupling may not be an issue (ie. standard libraries)

Due to the iterative process of software development this is often not the case and coupled code should be avoided.

High Cohesion

Use Cases:

Assign responsibilities to classes so that cohesion remains high.

If introducing too many responsibilities to a class that are not related, consider splitting out into helper classes.

Evaluative principle, aids with comprehension and maintainability similar to low coupling.

Contraindications:

Low cohesion is occasionally needed to meet strict non-functional requirements

- E.g a larger (and thus less cohesive class) can be more performant because less data is being transferred to different classes/systems

Protected Variation

Use Cases:

When there is a point of potential variation (ie. different databases) create a stable interface around that point.

Contraindications:

Cost of development time can outweigh potential future benefits.

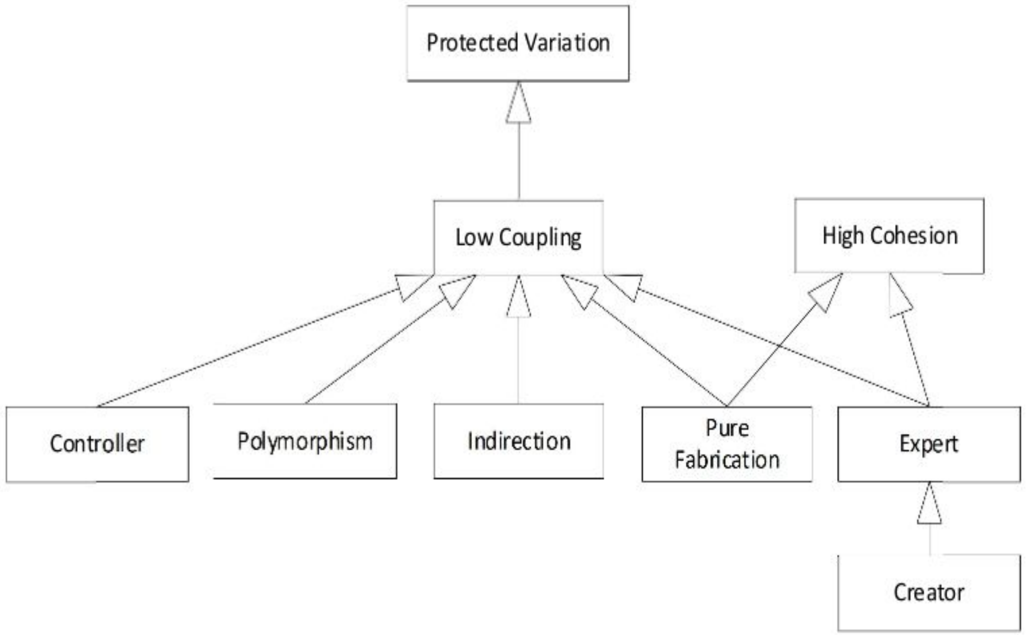
Could be cheaper to rework a brittle component, rather than design a robust one.

Using GRASP

Make sure to properly **justify** your answers and the effect that your answer has

E.g. don't just say there's high coupling, high coupling between what classes?

Don't just say a class is an expert, what specifically does it know?



How to answer questions

We want to introduce multiple dice to Monopoly game, how do we do it?

Analysis	MonopolyGame class contains all the info for rolling dice as it contains numberOfDice and numberOfDiceSides.
Answer	By Expert, we should assign this responsibility to MonopolyGame.
Effect	However, this lowers cohesion as MonopolyGame has to manage both dice rolls and is a controller for the game.
Analysis	The functionality cannot be assigned to an existing class in the domain model and adding it to an existing class would lower cohesion.
Answer	So, by Pure Fabrication, we should create a new class (cup) for this responsibility.
Effect	This maintains cohesion for MonopolyGame so it’s only focused on being a controller for the game and the cohesion forcup is high because it’s only focused on maintaining dice rolls

Gang of Four

GoF - Creational

- Factory
- Singleton

GoF - Factory

Use Cases:

When object creation is complex (many conditionals, complicated parameters) use pure fabrication to delegate responsibility to a factory class.

This keeps object creation cohesive and hides the complexity from the classes that use the objects.

Contraindications:

Use object constructors instead when creation is simple.

If there are many variations on a similar theme - an abstract factory that creates factories of a specific type can be useful.

GoF - Singleton

Use Cases:

When it needs to be ensured that only one instance of a class is created, or that class needs to be used in many places and would introduce too much coupling -

create a singleton with a static method to create/return the object.

Contraindications:

Using singleton will create global unmitigated access to the class which may not be desired.

May not be very easy to test/modify later on in the development cycle (brittle code)

GoF - Structural I

Adapter

Problem:
How to resolve incompatible interfaces, or provide a stable interface to similar (but different) components?

Solution:
Convert the original interface of a component into another interface, through an intermediate adapter object.

GRASP:
Protected variation, polymorphism, indirection

Adapter vs Facade
Adapter involves varying external systems and wants polymorphism. Facade is about one subsystem

Facade

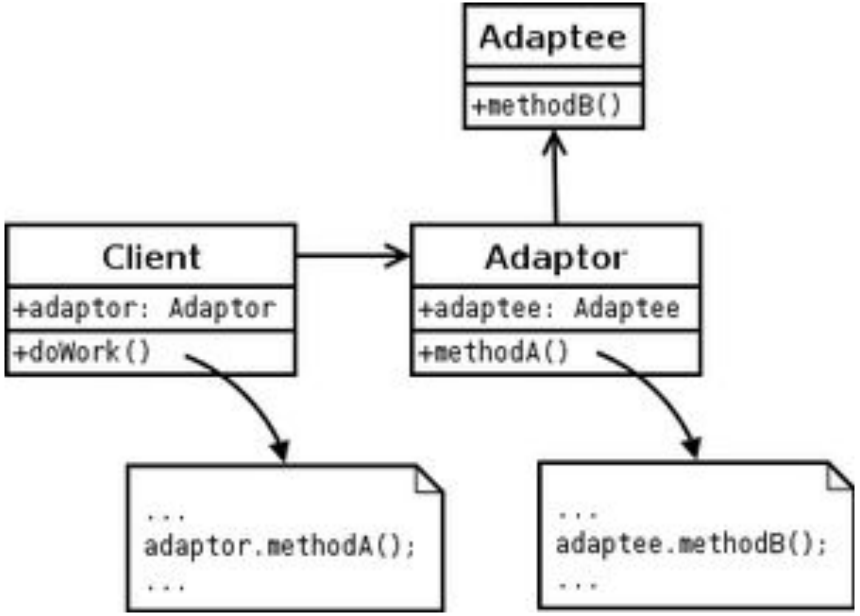
Problem:
We have a complex system or subsystem with functions that we want to access. Maybe it is tightly coupled within itself. Subsystem may change in future.

Solution:
Define a single access point to the subsystem - a facade object wraps the subsystem. Is a single unified interface. Handles collaboration with subsystem components.

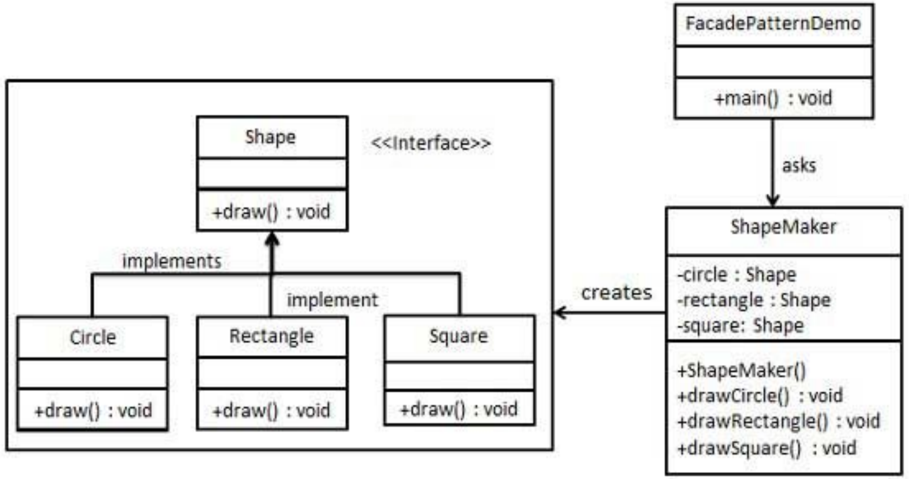
Facade vs Controller
Facade actually handles complexity! Controller is UI/input driven & delegates complexity.

GoF - Structural I

Adapter



Facade



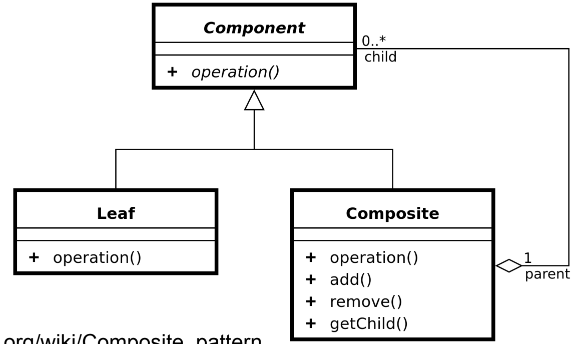
https://www.tutorialspoint.com/design_pattern/images/facade_pattern_uml_diagram.jpg

GoF - Structural II

Composite

Problem:
How to treat a group of objects the same way (polymorphically) as atomic objects?

Solution:
Define classes for a composite and atomic objects, and have them both implement the same interface.



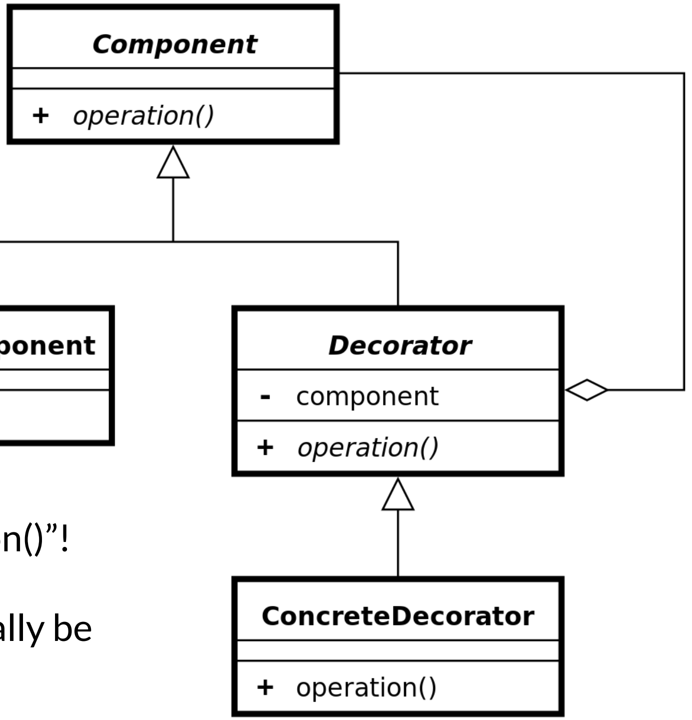
https://en.wikipedia.org/wiki/Composite_pattern

Decorator

Problem:
How to dynamically add behaviour to individual objects, without affecting the behaviour of other objects from the same class.

Solution:
Encapsulate original concrete object inside an abstract wrapper interface. Let decorators (with dynamic behaviours) inherit from this interface. The interface will then use recursive composition to allow an unlimited number of decorator “layers” to be added to each core object

GoF - Structural II - Decorator



Note both concrete classes share “operation()”!

Enables decorated objects to polymorphically be used the same way as when undecorated. Different behaviour though

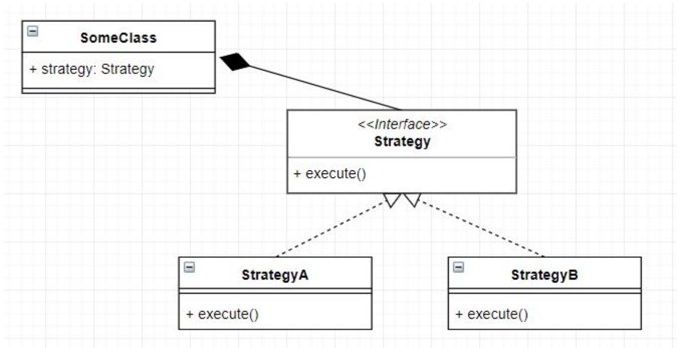
https://en.wikipedia.org/wiki/Decorator_pattern

GoF - Behavioural

Strategy

Problem:
We want to vary some behavioural element. How to design for varying, but related algorithms or policies? How to design for the ability to change these algorithms?

Solution:
Define each algorithm/policy/strategy in separate classes with a common interface. Pass in context objects.

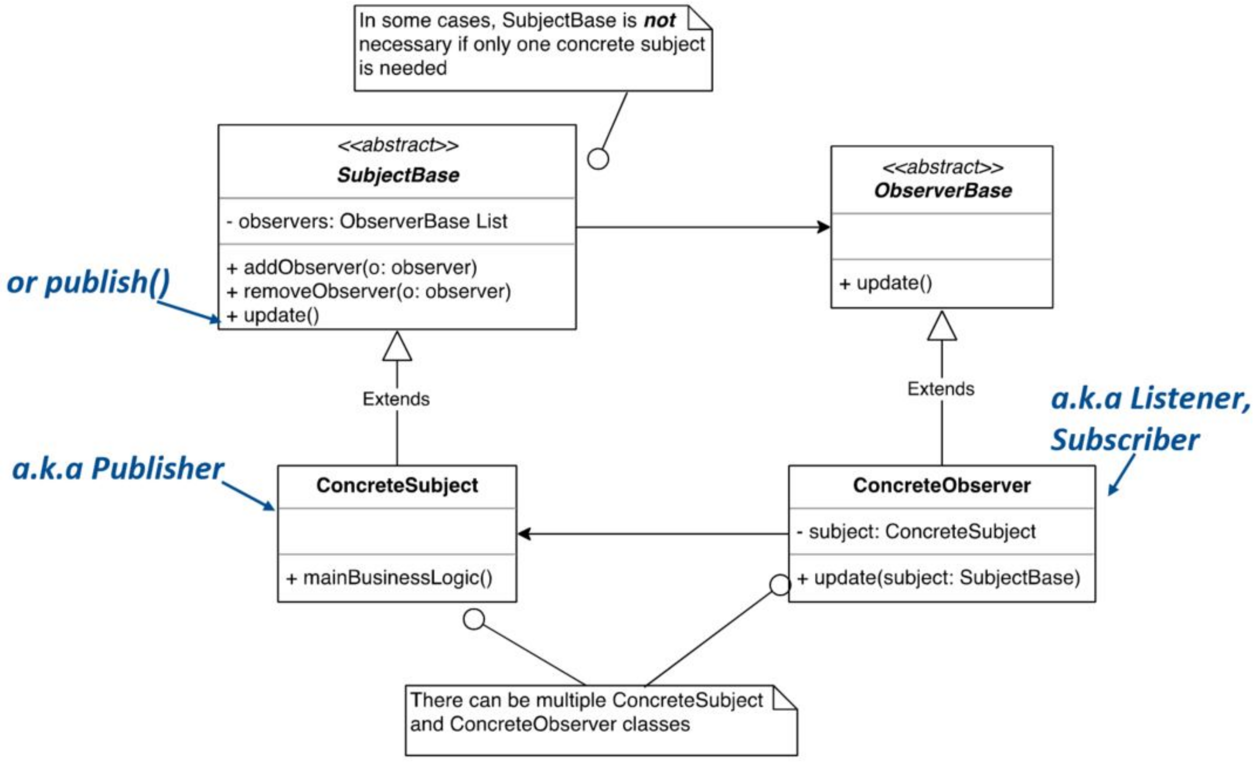


Observer

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.

GoF - Behavioural - Observer

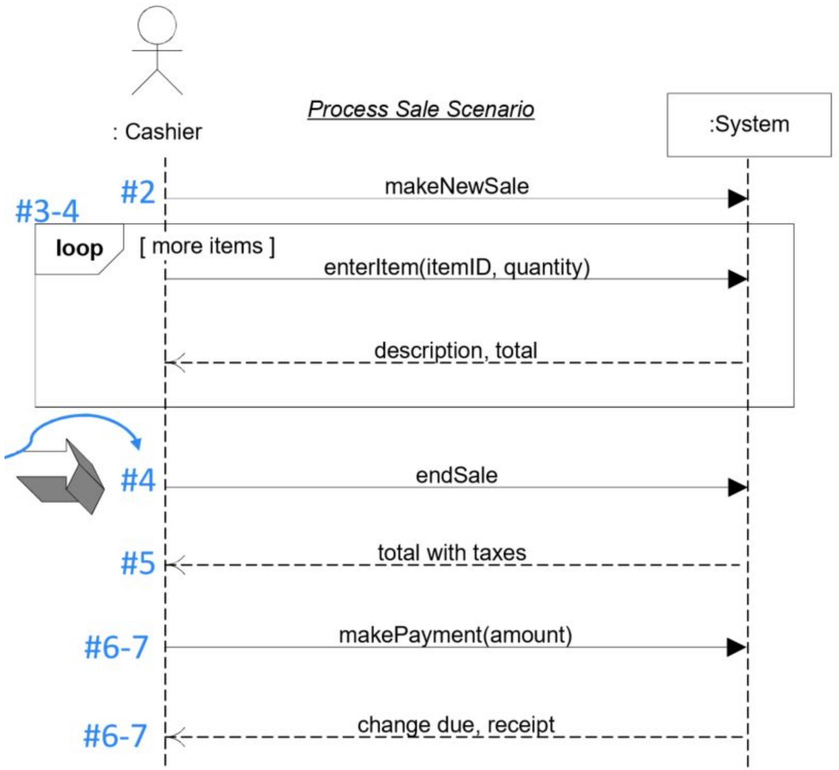

In some cases, SubjectBase is *not* necessary if only one concrete subject is needed
<<abstract>>
SubjectBase
- observers: ObserverBase List
+ addObserver(o: observer)
+ removeObserver(o: observer)
+ update()
or publish()
<<abstract>>
ObserverBase
+ update()
Extends
Extends
a.k.a Listener, Subscriber
a.k.a Publisher
ConcreteSubject
+ mainBusinessLogic()
ConcreteObserver
- subject: ConcreteSubject
+ update(subject: SubjectBase)
There can be multiple ConcreteSubject and ConcreteObserver classes

Example - System Sequence Diagram

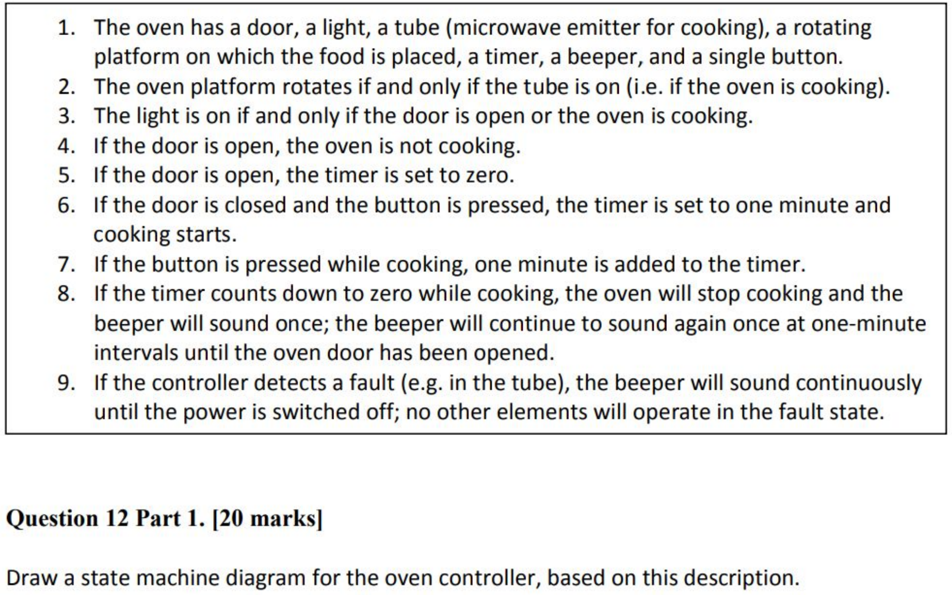
This is a scenario (similar to a test case) for a use case

<u>Simple cash-only *Process Sale* scenario:</u>

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
...



Example - State Machine



1. The oven has a door, a light, a tube (microwave emitter for cooking), a rotating platform on which the food is placed, a timer, a beeper, and a single button.
2. The oven platform rotates if and only if the tube is on (i.e. if the oven is cooking).
3. The light is on if and only if the door is open or the oven is cooking.
4. If the door is open, the oven is not cooking.
5. If the door is open, the timer is set to zero.
6. If the door is closed and the button is pressed, the timer is set to one minute and cooking starts.
7. If the button is pressed while cooking, one minute is added to the timer.
8. If the timer counts down to zero while cooking, the oven will stop cooking and the beeper will sound once; the beeper will continue to sound again once at one-minute intervals until the oven door has been opened.
9. If the controller detects a fault (e.g. in the tube), the beeper will sound continuously until the power is switched off; no other elements will operate in the fault state.

Question 12 Part 1. [20 marks]

Draw a state machine diagram for the oven controller, based on this description.

https://course-work.unimelb.edu.au/handle/11343/212976

Exam tips

- Put pen to paper → you will probably iterate and improve as you draw
- Collate all subject material to use in the exam (e.g. merge PDF)
- **Use the “notational guides” for UML diagramming on the LMS!**
- Have a backup plan for uploading images. Test the process before you sit the exam

- Our experiences in the exam