Design Defects & Restructuring

Week 7: 22 Oct 22

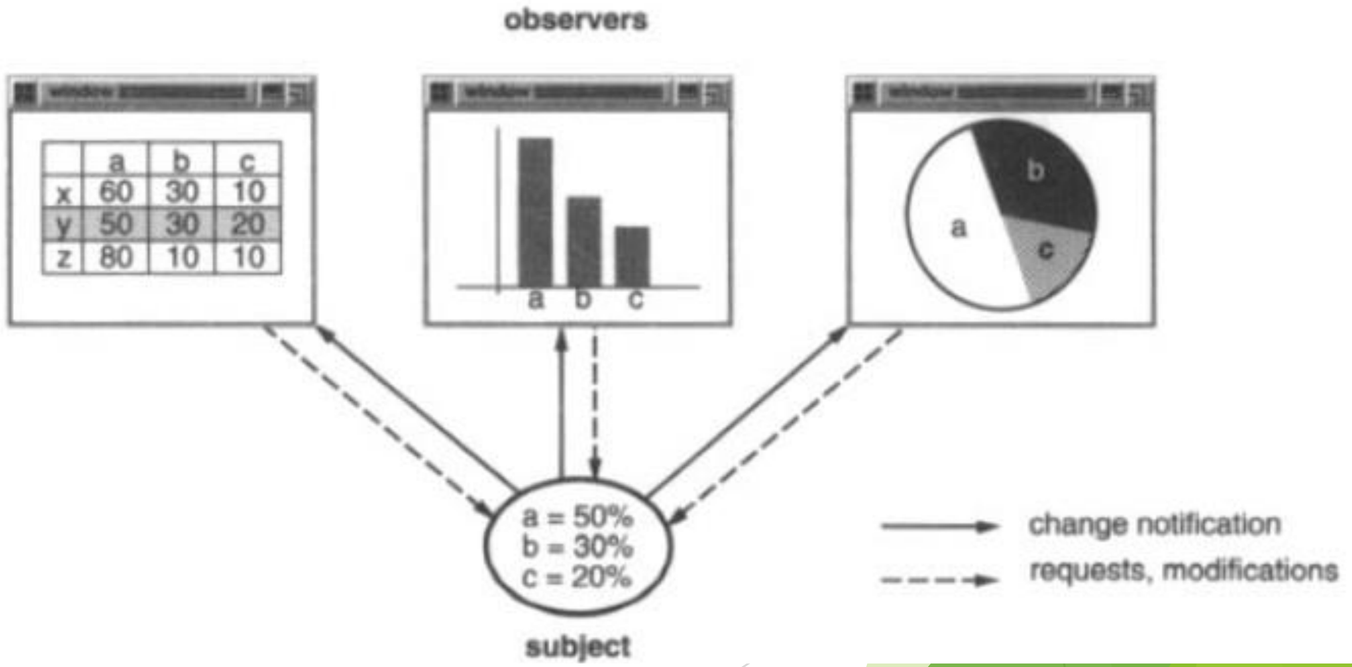
Rahim Hasnani

Observer Pattern from GOF

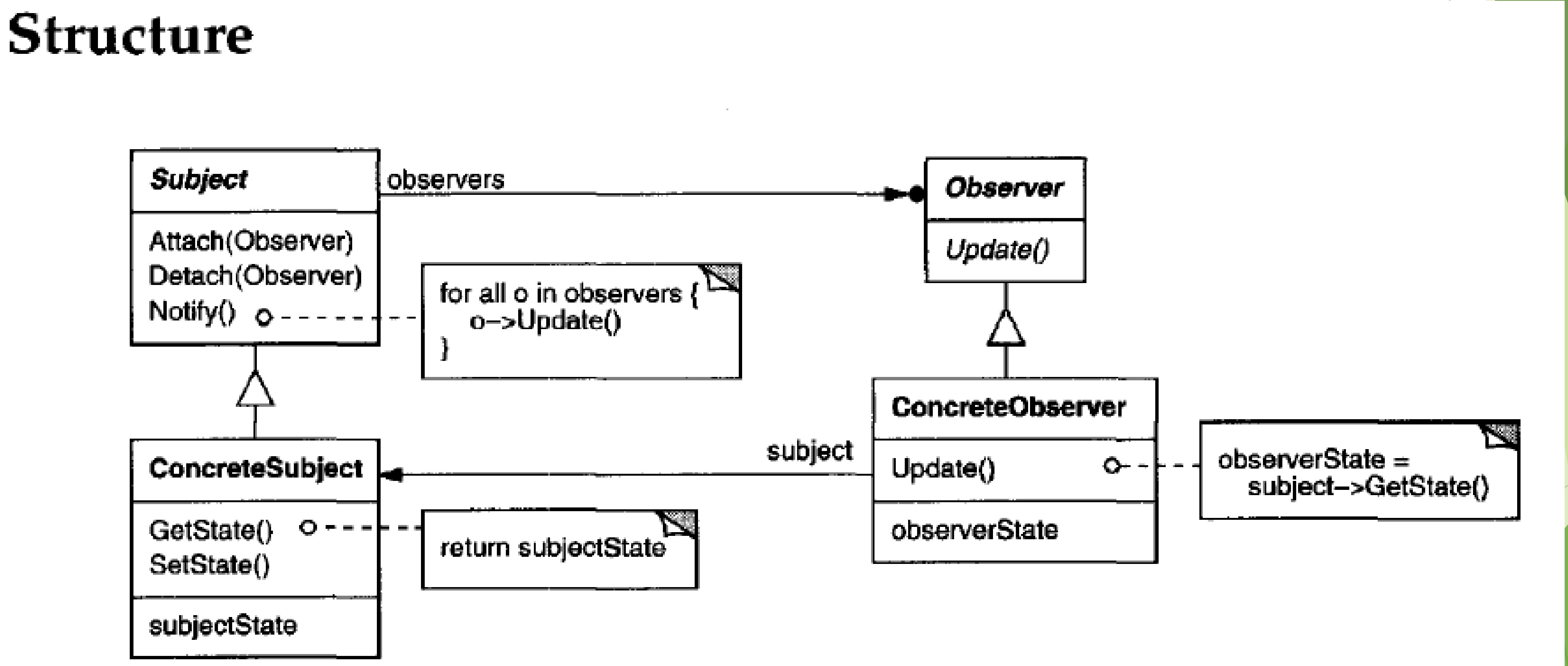
- Intent
  - Define a one-to-many dependency between objects so that when one objectchanges state, all its dependents are notified and updated automatically.
- Also Known As
  - Dependents, Publish-Subscribe
- Motivation
  - A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
  - For example in a GUI based system, classes defining application data and presentations ca n be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately and vice versa.

Observer Pattern from GOF

- The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependent observe rs. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.
- This kind of interaction is also known as publish-subscribe. The subject is the publisher o f notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.



Observer Pattern from GOF



Refactoring - Definition

- As a noun:
  - “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.”
  - Example: Extract Function
- As a verb:
  - “to restructure software by applying a series of refactorings without changing its observable behavior.”
  - Example: ‘I might spend a couple of hours refactoring, during which I would apply a few dozen individual refactorings.’
- Key concepts:
  - Small changes
  - Changes/steps are behavior-preserving

Why Refactor

- Refactoring improves the design of software
  - As we add more code, it tends to decay – so need to periodically refactor
- Refactoring makes software easier to understand
  - Refactored code is more readable – easy to understand by future programmer
- Refactoring helps me find bugs
  - When carrying out refactoring, deep understanding of code is sought, which helps find bugs
- Refactoring helps me program faster (!!)
  - How? At first writing code without refactoring would be faster, but as we put more code into existing code base, it becomes harder to do it efficiently without refactoring first

When to Refactor

- Preparatory Refactoring – Making it easier to Add a feature
- Comprehension Refactoring: Making code easier to understand
- Litter-Pickup Refactoring
- Planned and Opportunistic Refactoring
- Long-Term Refactoring
- Refactoring in a Code Review

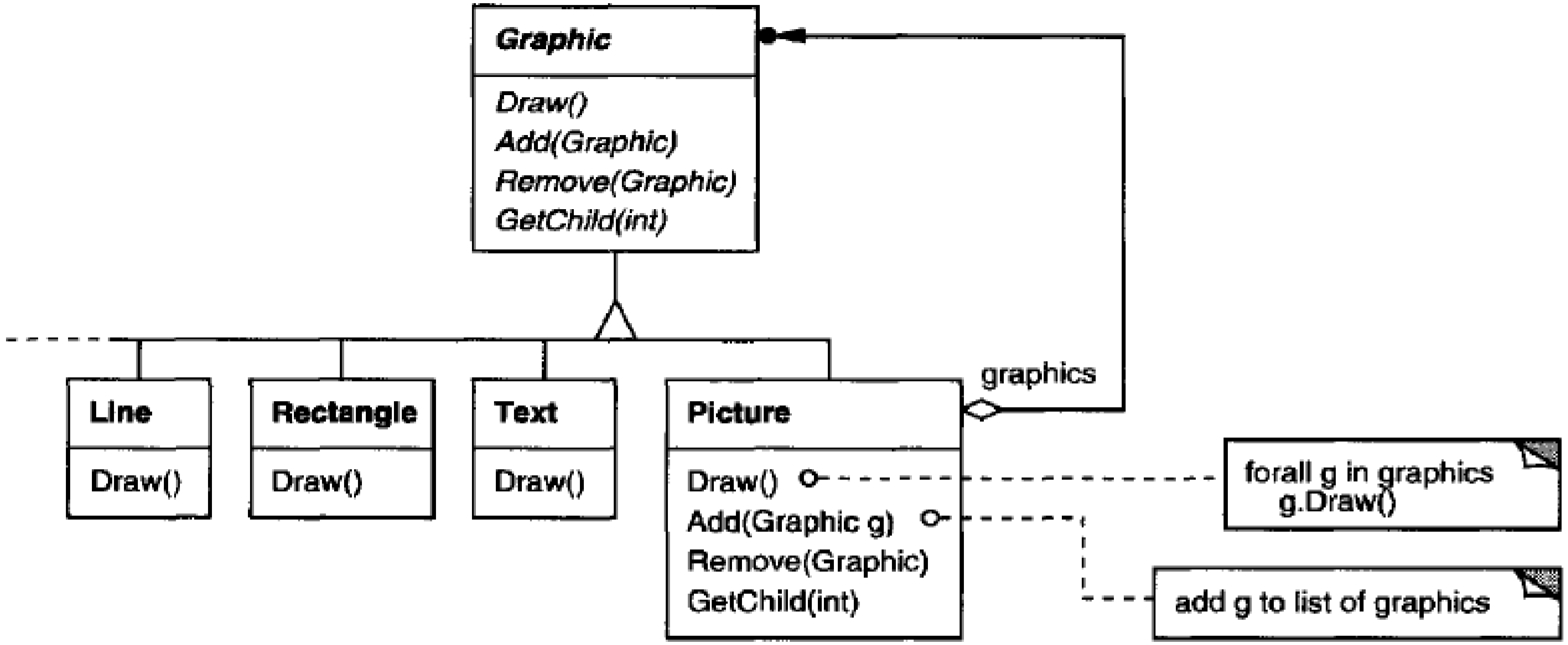
- When to Not Refactor

Problems with Refactoring

- Slowing Down New Features
- Code Ownership
- Branches
- Testing
- Legacy Code
- Databases

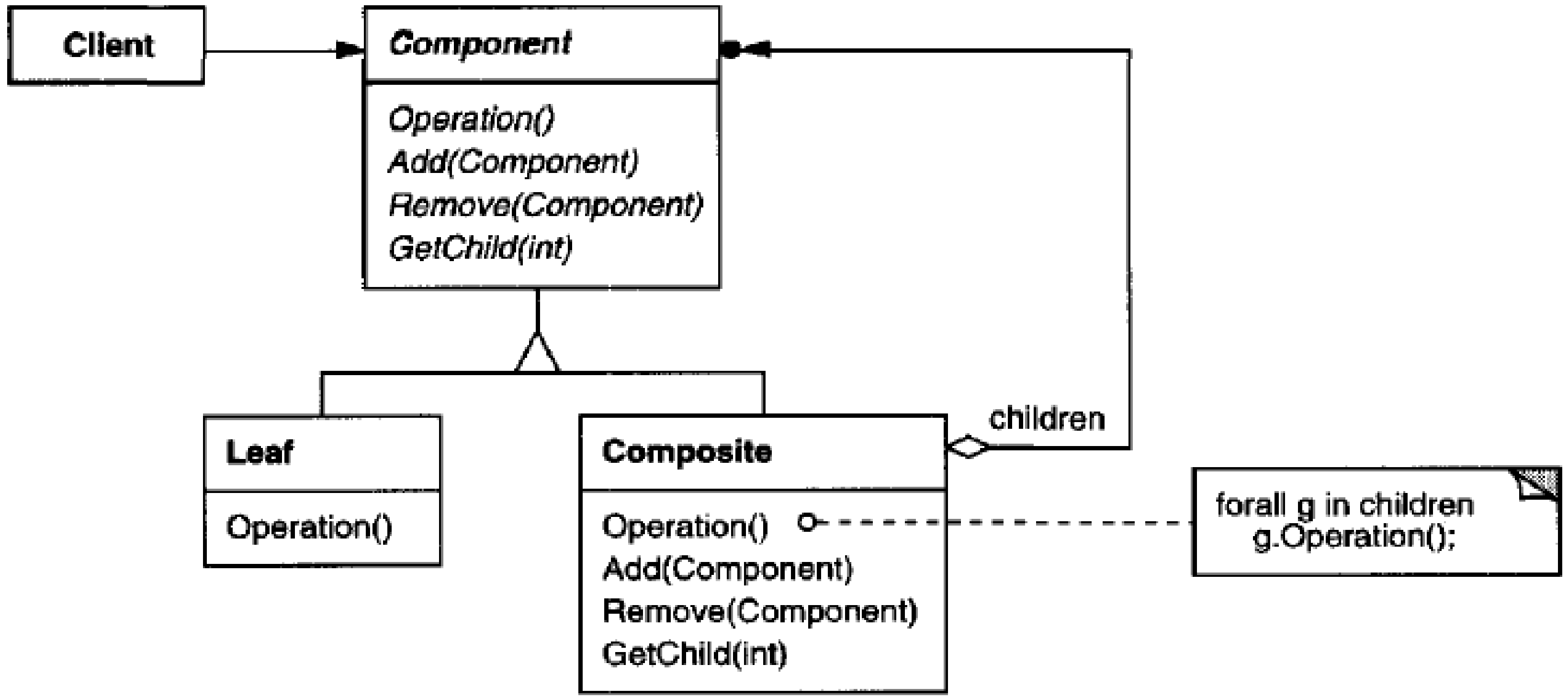
Design Pattern: Composite

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Motivation
  - Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.
  - But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
add g to list of graphics

Design Pattern: Composite

- Applicability
  - Use the Composite pattern when
    - you want to represent part-whole hierarchies of objects
    - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
- Structure



Design Pattern: Composite

- Participants
  - Component (Graphic)
    - declares th e interfa ce for obje cts in the composition.
    - implements default behavior for the interface common to all cl asses, as appropriate.
    - declares an interface for accessing and managing its child components.
    - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
  - Leaf (Rectangle, Line, Text, etc.)
    - represents leaf objects in the composition. A leaf has no children.
    - defines behavior for primitive objects in the composition.
  - Composite (Picture)
    - defines behavior for components having children.
    - stores child components.
    - implements child-related operations in the Component interface.
  - Client
    - manipulates objects in the composition through the Component interface.

Design Pattern: Composite

- Collaborations
  - Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.
- Consequences

  The composite pattern

  - defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
  - makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition
  - makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.
  - can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Exercise

- **Problem:** Create a program that reads new records from the database based on certain filter criteria and generates a file in a specified fixed length format on a location.
- **First Version:**
  - A console based application is made
  - The following is set through configuration (what is configuration??)
    - Database connection string
    - Query that provides the new records
    - Output file folder location
  - A single file code that carries out following:
    - Establishes connection with database. Throws error and exits if it cant connect
    - Runs the query and stores the result in memory. Throws error and exits if there is error
    - Verifies that query returns the correct number of column, otherwise reports error and exits.
    - Iterates over each row of the result and carries out validation (certain fields need to be present) and ignores those rows that do not fulfill the validation criteria. If all is well with a row, it converts the row into the required fixed format record for writing to file.
    - Opens file for writing and write all the converted records

Question 1:

- What are the design goals of client?
- What should be the design goals of the architect/designer to create a robust program?

Question 2

- Provide a design