Object – Oriented Design with UML and Java

Part II: Design Process

Version 2023

High-Level Process

- Establish a business case and hire a team.
- Model the problem domain
  - Develop an ***analysis model***
- Develop ***use cases*** / ***user stories***
- Model the solution domain
  - Develop a ***design model*** (pass one)
  - Ensure that the high-level ***architecture*** is appropriate
- Implement, test, release...
- ***Iterate…***
  - Within each step, and across all of them
  - Steps are only approximately sequential

Analysis vs. Design

- Analysis describes ***what***
  - An existing business process
  - An existing computing infrastructure
  - The requirements for a solution
- Design describes ***how***
  - A new set of screens
  - A new application
- The distinction is not always clear-cut
  - The customer says “The new system must do x, y and z.”
  - Modeling x, y, and z is analysis
  - Modeling the mechanisms to accomplish x, y, and z is design

Architecture – Design - Implementation

- **Architecture**
  - Programming Languages, Frameworks, Infrastructure, Cloud
  - Security, Legacy Systems, TLAs, NFRs, APIs
  - Databases, Scalability, Resilience, Modularity, Strategy
  - Vendor lock-in, Technology Stack, Team Topology
  - Plan for Change / Iterative Evolution / Continuous Delivery / Loose Coupling
- **Design**
  - Reduce Complexity, Separate Concerns, Reuse
  - Delegation, Generalization, Interfaces, Layers
  - User Interface, Transactions, Modules, Patterns
- **Implementation**
  - Testing, Data Structures, Algorithms, Thread Safety
  - Encapsulation, Idioms, Memory Management

Use Cases / User Stories

Organize ***Functional Requirements*** in a way meaningful to users, describing their interactions with the system in support of their goal(s).

- ***Use Cases*** are descriptions of system processes, workflows, or scenarios, designed to ensure that the system behavior is what the users require.
- ***Actors*** are the different user roles in a system, both people & machines. Examples: general user, system administrator, the payroll subsystem.
- The model describes ***what*** the system should do without specifying ***how*** it should be done.
- A Use Case model is NOT a complete requirements specification.
- Consider ***non-functional*** requirements as well (e.g. performance).
- Use Cases imply requirements through the stories they tell.
- A complete set of Use Cases helps to scope a project.
- Think about the ***value*** being delivered to the system's users.

Model the Problem Domain

- ***Analyze*** the requirements & use cases enough to get started.
- Maybe make a User Interface ***mockup*** with a tool.
- Parties, Places and Things can all have Roles. Model these.
  - A person can be a customer in one scenario and a vendor in another.
- Consider the business process ***moment intervals.***
  - What interactions need to be remembered?
  - Are they sequential in time?
- For each candidate object, propose a class.
  - Eliminate duplicates and things outside the system (e.g., the user).
  - An adjective may suggest an attribute or a subclass.
- Develop interaction models that support the use case scenarios.
- Identify collaborations & relationships.
- Identify responsibilities for classes.
- Break up big, complex classes; consolidate trivial classes.

Model the Solution

- ***Design*** a solution to support what you've analyzed.
- Develop an overall system ***architecture***.
- Strive for the simplest solution that will work.
- Avoid over-designing; design only enough for what you know you need.
- Create **UML** models (with pertinent detail).
- Assign operations to classes that will implement the class' responsibilities.
- Identify attributes / associations / inheritance hierarchies.
- Refine existing interaction diagrams and define new ones.
- Study (code) design and (system) architecture patterns.
- Ask yourself: What infrastructure is needed in order to make the software easier to extend in the future? Is it easy to build? Will it make things simpler now?
- ***Iterate*****…** to develop the simplest solution that will fulfill all of the requirements, while being flexible for future requirements and reuse.
- Realize that you will have more opportunities to refactor later on…

Build and Test

- Implementation involves fine-grained design.
- Implementation issues may suggest or require coarser-grained re-design.
- Design test cases before you code, if possible. Maintain test code.
- In this course with Java, we will use ***Junit*** for unit testing.
- In the wild, use automation to test every time you check code into ***GitHub***.
- Implement a sound ***exception handling*** and ***logging*** strategy right away.
- Use meaningfulVariableNames & write good comments for tricky details.
- Constantly focus on the overall architecture, adding detail & functionality while occasionally ***refactoring*** to simplify or adapt to changing requirements.
- Keep clear distinctions between layers & subsystems.
- Strive for simplicity, generalization, reuse & style.

Google ***Test-Driven Development*** and ***Continuous Integration***.

Development Lifecycles

Waterfall

- Linear and sequential phases
  - Analysis
  - Design
  - Code & Test
  - Integration
  - System Test / User Acceptance Test
  - Deployment
- Each phase is visited exactly once.
- Can be predictable with low risk IF requirements are clear and stable

Incremental & Iterative

- Phases are repeated to continuously improve the product
- Agile methodologies such as Scrum, Kanban, and Lean
- Quickly adapt to changes in requirements

Iterative & Incremental Development

- Analyze and specify only as much as you understand. Defer the rest until you have built what you understand, and understand what you have built.
- Get a working system as soon as possible... but not too fast!
- Be ***architecture-centric***, refactoring design elements as you go along to make them simpler or more flexible, or both.
- Do not be afraid to break working code to ***refactor*** your design if the improved design is worth the effort.
- ***Test*** relentlessly, with a great collection of unit and functional tests.
- Good process helps create better code.
- These activities are interlaced, not strictly sequential.
- Value working software with continuous user input over thick documents and contracts.
- Avoid analysis paralysis.

Refactoring

- Refactoring is the process by which existing portions of code are carefully changed, without adding any new functionality, to become simpler, more general, more flexible, more reusable, easier to understand, easier to maintain, smaller, faster, ...
- Refactoring code involves breaking code that already works, so it is counter-intuitive to people who believe in the old adage “if it works, don’t fix it!” Refactoring can be risky! However, it is often worth the effort.
- Refactoring is *disciplined* code evolution. Make small changes, one at a time, to minimize the chance of introducing bugs.
- A good set of unit and functional tests will give you confidence to do more refactoring, with less risk.

eXtreme Programming

- onsite customer / user
- visual modeling
- pair programming
- continuous integration - relentless testing
- refactor mercilessly
- you aren’t going to need it
- frequent iterative releases
- planning game w/ user stories - customers - developers – iterations
- architecture - patterns - reusable infrastructure
- collective code ownership - coding standards
- do the simplest thing that could possibly work
- they’re just rules

Responsibility Driven Design

A class’ responsibilities define its public interface.

- Walk through the use cases, looking for interactions between objects.
- Where there are interactions, identify client-server relationships.
- Every interaction implies the server has a responsibility.
- Phrase the responsibility from the point of view of the client. This becomes the name of the server’s behavior (function, method or operation) that supports the responsibility.
- A large set of responsibilities implies the class should ***delegate*** some work.
- If a client requires a service that does not belong to an existing class, create a new class.

Responsibilities & Collaborations

Responsibilities:

- All the services the instances of a class provide to other objects.
- All instances of a class have the same responsibilities.
- Responsibilities include:
  - Performing actions (methods, behaviors).
  - Maintaining and providing knowledge (state, attributes).
  - Enforcing constraints.

Collaborations:

- Client / Server relationships “in the small.”
- A collaboration is one class calling on another (sending a message) to help fulfill a responsibility.

Example: A Microwave Oven Simulator

Requirements Statement:

A microwave oven heats food by bombarding it with microwaves. The hungry user pushes buttons on a keypad to tell the oven how long to heat the food, and at what power level. The keypad also has a button for starting the oven. When the timer times out, it stops the oven and rings the bell. The microwave will also stop if the door opens. Assume that the microwave generator works by controlling a Klystron with a one second duty cycle.

- *Find the classes...*

Microwave Oven - Use Cases

1) Set Time
   - User inputs a Number, then presses the Set Time Button.
   - System updates Display for time left.
   - System sets Timer.

2) Set Power Level
   - User inputs a Number, then presses the Set Power Level Button.
   - If the number is not in (1-10) then do nothing.
   - System updates Display for power level.
   - System sets microwave Generator.

3) Open Door
   - Stop Timer from counting down.
   - Disable Generator from generating microwaves.

Microwave Oven Use Cases

4) Close Door

    Enable microwave Generator.

5) Push Start button

    If Door closed:

    Begin counting down.

    Generate microwaves, while time left > 0.

    Keep Display up to date with time left.

    Stop and ring Bell when time = 0.

Microwave Oven – Classes (analysis)

- Oven
- Generator
- Timer
- Clock
- Keypad
- Start Button
- Door
- Bell
- Other Buttons:
  - Clear, [numbers] 0-9, Set Time, Set Power Level
- Klystron
- Display

Microwave Oven: Architecture

- There are three types of events in the user interface:
  - buttons being pushed.
  - the door being opened or closed.
  - timing events from a clock chip.
- We do not wish to continuously poll the Door to see if it is open or closed, nor the Timer to see how much time has gone by; an alternative is for the Oven to be notified when the Door is opened or closed, and when the Timer ticks.
- Use an ***event-driven***, rather than a ***polling*** architecture.
- Assume that the events of interest cause methods to be invoked on your classes. Don’t worry about how this works for now.

Microwave Oven: Architecture

- We wish for all of the components of the Oven (the Timer, the Keypad, etc.) to be reusable for an entire product family of ovens.
- The Button classes will not have responsibility for system control. We will keep the Button classes simple, with a single uniform interface function: push(). To have a reusable design, we do not want the buttons to know about all the components in the system.

Microwave Oven: Responsibilities

Oven:

- Know the state of the system.
- Delegate button commands to aggregated objects.
- Get notified when the Door gets opened or closed.
  - Disable Generator.
- Get notified when the Timer has counted down to 0.
  - Disable Generator.
  - Ring Bell.
- Get notified when the Timer has ticked.
  - Update Display.
  - Notify Generator.

Microwave Oven: Responsibilities

Microwave Generator:

- Know power level.
- Generate microwaves at power level, if enabled.
- Get notified of clock tick events.
  - » Activate the Klystron for a varying percentage of the microwave generation duty cycle, depending on the power level (integer between 1 and 10).

Timer:

- Know time left.
- Count down.
- Notify the Oven of the following events:
  - » The clock has ticked (every tenth of a second).
  - » The time left has reached 0.

Microwave Oven: Responsibilities

Display:
- Show time left.
- Show power level.
- Know and show current number being input into Keypad.

Keypad:
- Be a container class for the Buttons.

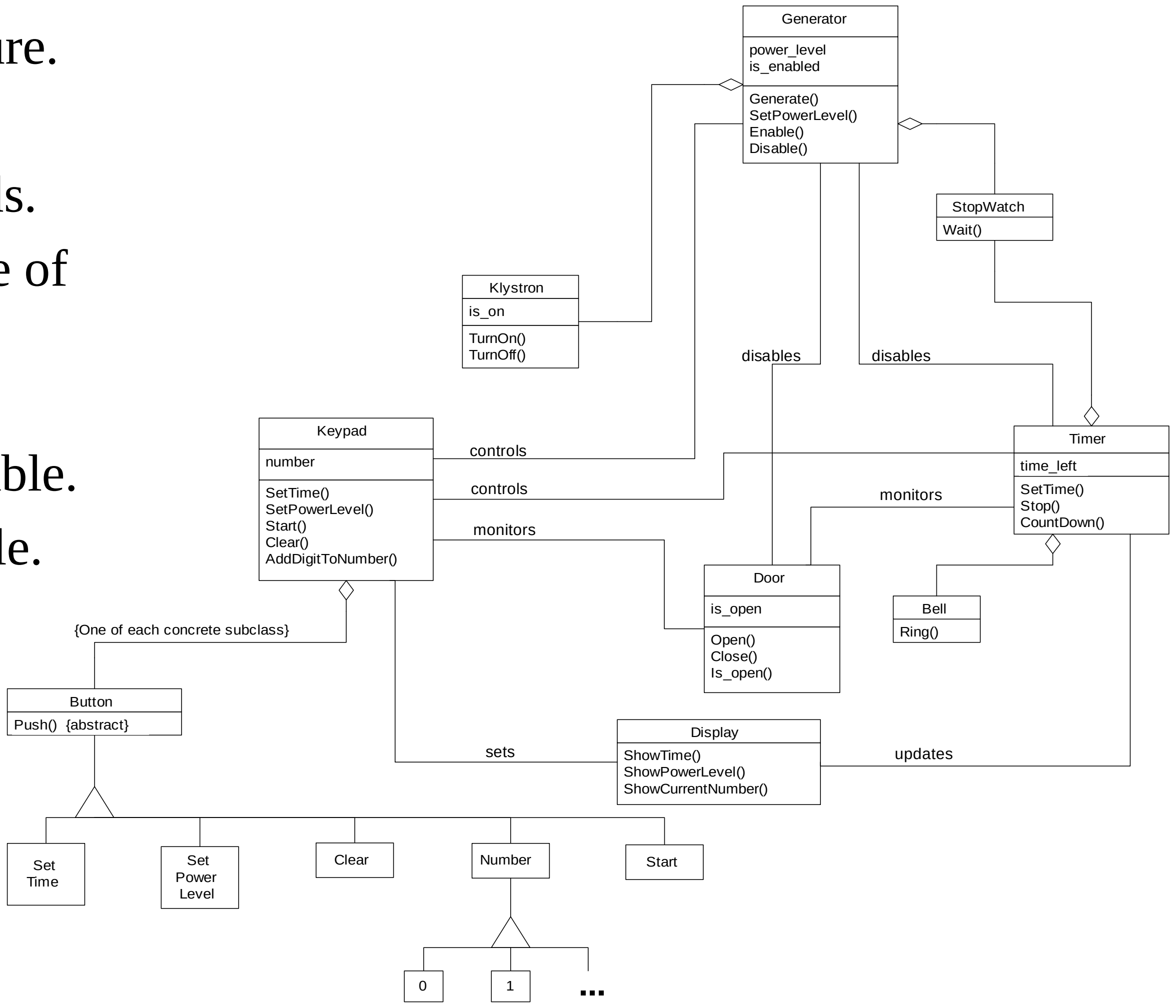
Door:
- Notify the Oven of open and close events.

Bell:
- Ring.

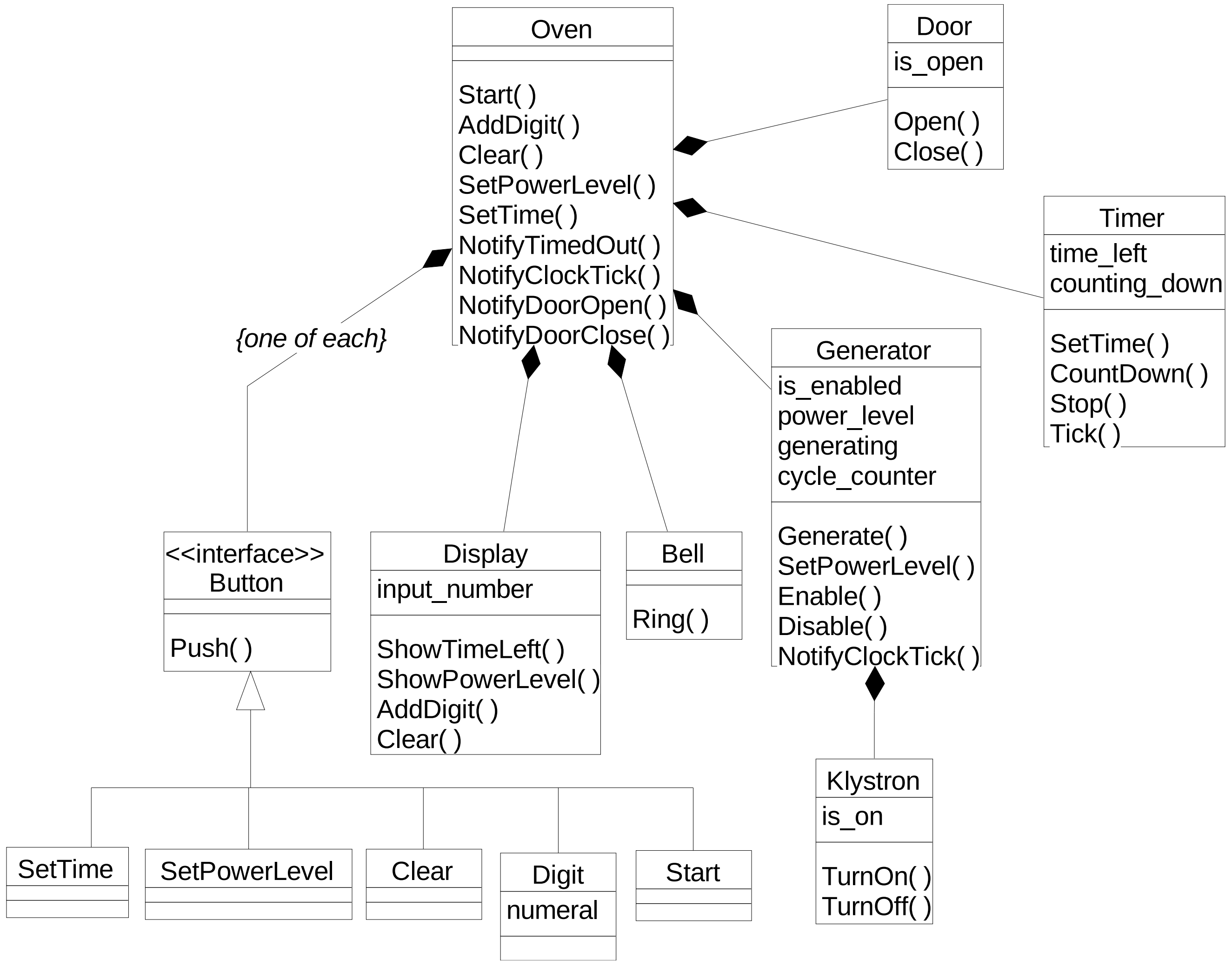
All Buttons:
- Get pushed; send a command to the Oven.

The Microwave Oven Class Diagram #1

- Polling architecture.
- Works fine, but...
- Requires 2 threads.
- Has a high degree of object coupling.
- Complicated.
- Keypad not reusable.
- Timer not reusable.
- ***REFACTOR*** ...



The Microwave Oven Class Diagram #2



Comparison of the 2 Microwave Oven models

- Model 2 has an Oven class (an example of the ***Mediator*** design pattern). This simplifies collaborations between components, which allows them to vary independently because they are now loosely coupled.
- Model 2 better resembles the ***event-driven*** problem.
- Model 2 does not have a Keypad class, as such a class would not have any responsibilities. In Model 1 however, it does have responsibilities.
- Models 1 & 2 also differ in the way they treat the 10 digits.
- Model 1 has a StopWatch class.
- Model 2 is both simpler and more flexible!

Design Pattern: *Mediator*

***Intent*:** Encapsulate the interaction(s) between a set of classes.

***Examples*:**

- The Oven
- The Sticks Game Referee.

***Applicability*:**

- Whenever the communication between classes gets complex.
- Whenever two or more interfaces must vary independently.
- Whenever information must flow between two classes, but neither class wishes to accommodate the other.

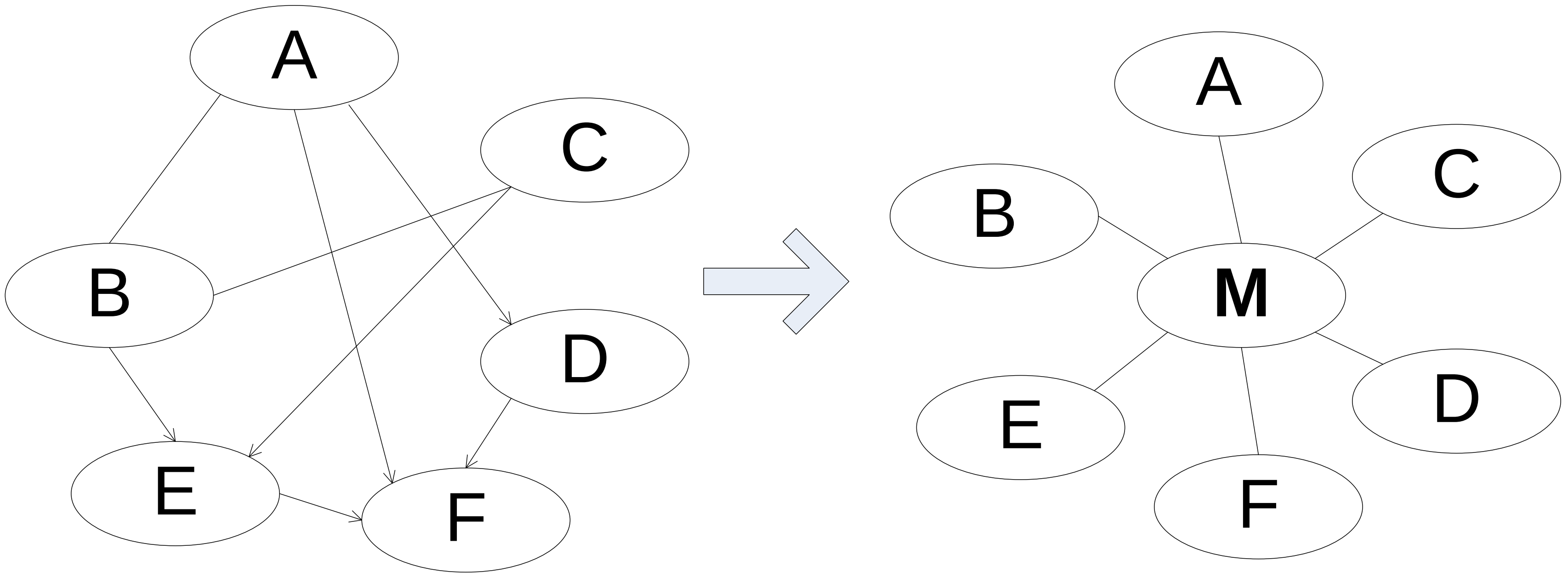
***Pros*:**

- Promotes loose coupling between objects.
- Simplifies complex collaboration diagrams.

***Cons*:**

- The Mediator itself can become quite complex.

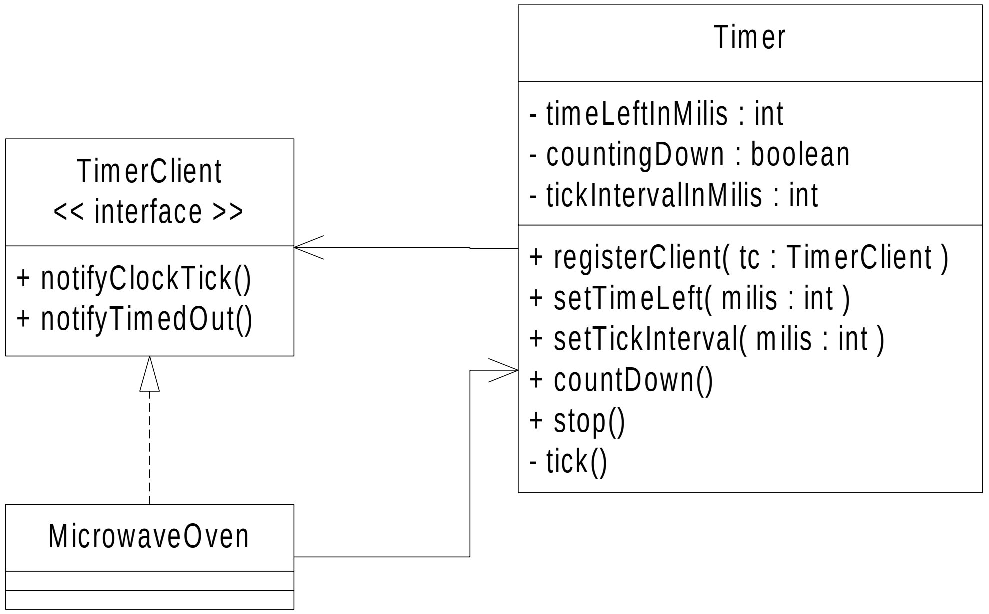
Mediator according to Graph Theory



N.B.: This diagram is not UML

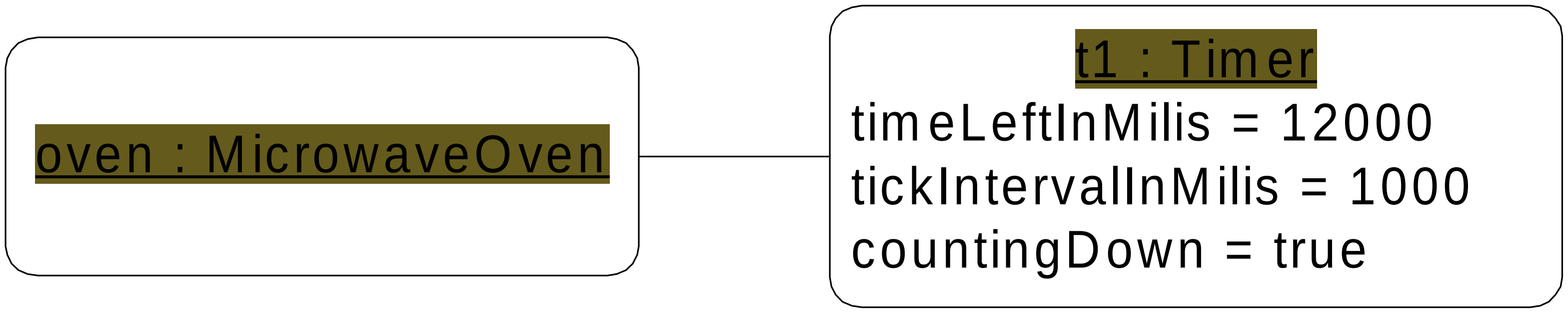
Reusable Timer Class Diagram

In order to make the Timer component ***reusable***, it must be ***decoupled*** from the context of the MicrowaveOven; the only way to do that is to invent a new ***interface*** and ***refactor*** the design...

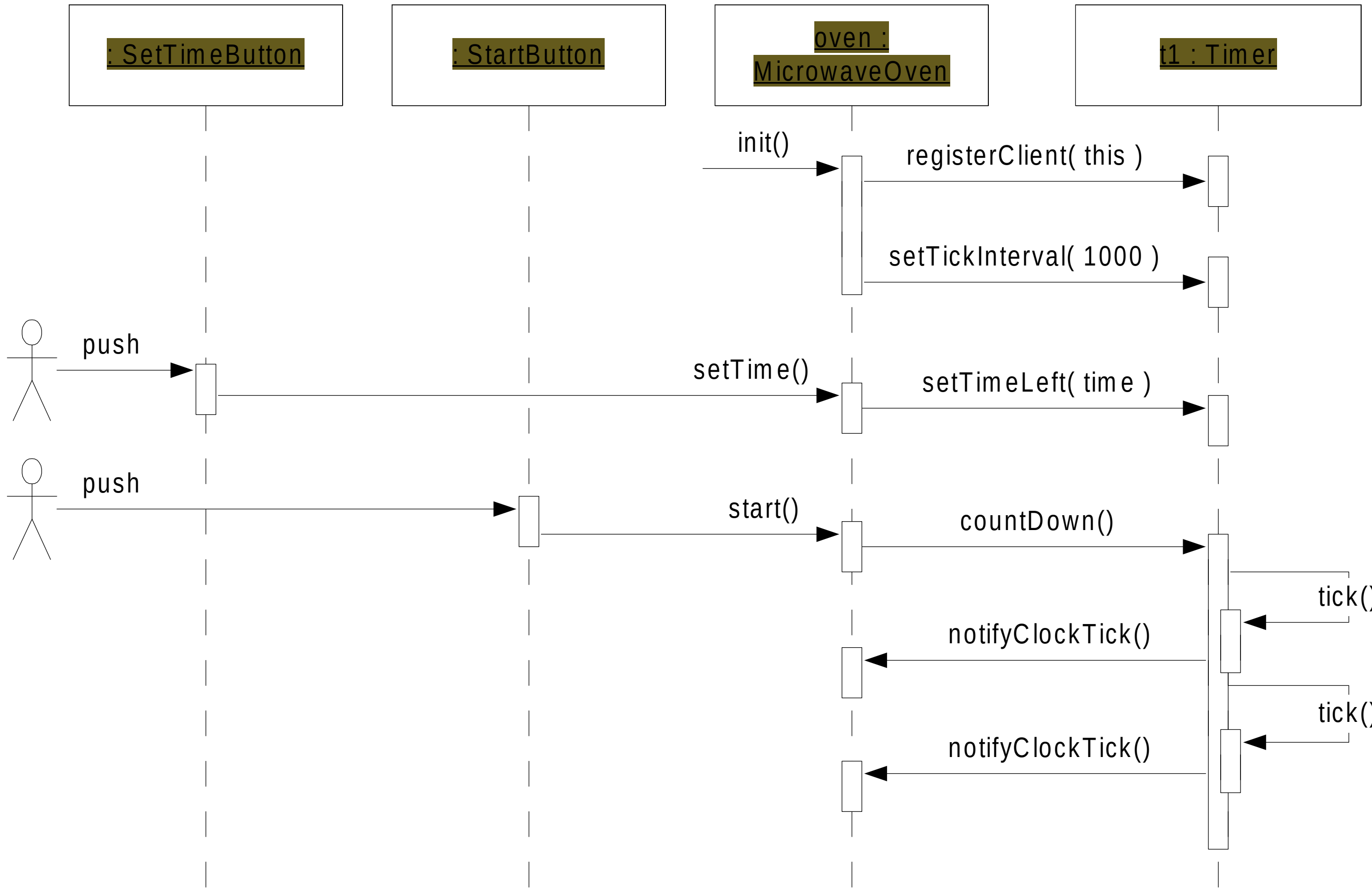


Reusable Timer Object Diagram

Notice that the TimerClient interface does not imply an object (because MicrowaveOven ***is-a-kind-of*** TimerClient).



Microwave Oven Sequence Diagram



Domains To Model

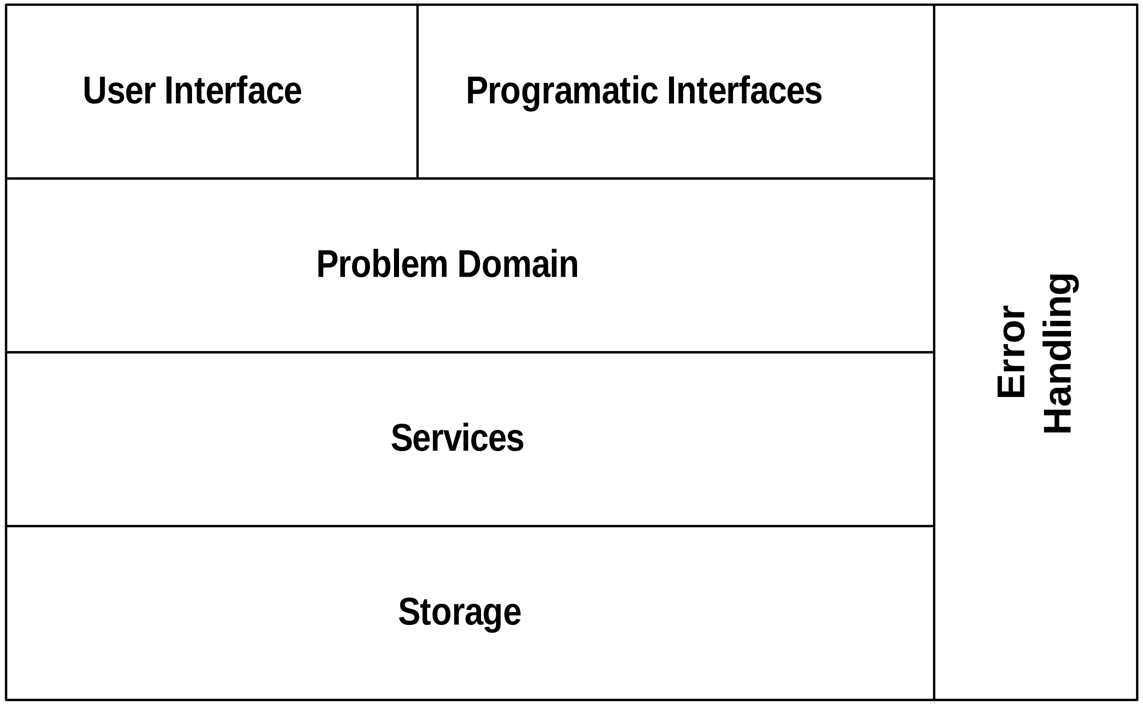
Every real-world system should be designed in several explicit domains.

- Business problem.
- Persistent storage of data.
- User interface.
- System context / network environment / security.
- Functional distribution.
- Solution ***infrastructure***.

Be careful not to mix responsibilities across domains.

- Separate concerns.

Layered Architecture Model



Model Physical & Conceptual Entities

OO Programming originated as a better way to think about writing *simulators* - programs intended to behave like physical processes.

Physical:

- Domain entities: automobiles, airplanes, …
- System entities: printers, modems, sensors, actuators, ...

Conceptual:

- Priority, Access privilege, Data format converter, Transaction manager, Memory manager, Data Structures
  - Graph / Tree / Queue / Stack / HashMap

Model Categories of Classes

- Animal
  - Dog
    - German Shepherd
    - Poodle
    - Mutt
  - Cat
- Magnetic Media
  - Disk
  - Tape

Natural categories often make good inheritance hierarchies.

Model Groups of Objects

- Game Layout
  - Rows
    - Sticks
- Corporation
  - Divisions
    - Departments
      - Employees

Groups of objects a modeled by ***composition*** and ***aggregation***.

Collections are special kinds of groups.

Model External Things You Must Call

Very important in the real world.

- Examples:
  - legacy code and data sources.
  - The operating system, if you call it.
  - vendor packages.
  - system interfaces.
- Identify the services provided.
- Create objects that are responsible for providing the services.
- Consider a ***proxy*** object for remote services to encapsulate the distributed communications.

Checking Your Classes

- Every class should have a clear name that sounds like a thing, not a function. ClassNames in Java should always begin with a capital letter and use MixedCaseLikeThis.
- Every class should have a clear purpose that applies to all of its subclasses.
- Classes that are similar but not identical might suggest inheritance, or possibly delegation to a third, shared class.
- Classes should only overlap if they have an inheritance relationship, and then one class should completely overlap the other.
  - Otherwise, classes should ***separate their concerns***.
- Classes may collaborate to fulfill their responsibilities.
- Classes should have semantically related attributes and methods.

Relationship Types

- Composition
  - B ***is part of*** A
  - A ***contains*** B
  - A ***has a collection of*** Bs
- Subclass / Superclass
  - A ***is a kind of*** B
  - A ***is a specialization of*** B
  - A ***behaves like*** B
- Collaborative
  - A ***delegates to*** B
  - A ***needs help from*** B
  - A and B are ***peers***.

Assigning Responsibilities

- Think about how the program will actually work.
- State responsibilities as generally as possible.
- Keep information about one thing in one place.
- Responsibilities can be shared or delegated.
- A class with no responsibilities is likely superfluous.
- Refactor your design whenever it gets too complicated.
- Break up big, complex classes.
- Centralized “intelligence” is inflexible.
- Avoid complexity in the graph of interacting objects.
- Poor choices lead to fragile systems.

“GRASP” Patterns

General **R**esponsibility **A**ssignment **S**oftware **P**atterns:

- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Controller

“The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology.”

- Larman, Craig. *Applying UML and Patterns - Third Edition*
- GRASP is a learning aid.

Information Expert

- Assign a responsibility to the class that has the information necessary to fulfill it. "Partial experts" collaborate.
- This is the most basic responsibility assignment principle.
- If you find yourself using many *setters* and *getters*, you may have violated this principle.
- Examples:
  - In the Sticks game, which class should have the responsibility of displaying the character for a stick?
  - In the Video Store, which class should have the responsibility for knowing if a video is overdue?
  - In the Sticks game, which class knows when the game is over?
  - Temperature converter.

Creator

- Every object must be created somewhere.
- Consider making a class responsible for creating an object if:
  - It has the information needed to initialize the object.
  - It will be the primary client of the object.
  - It is an inventory of objects of that type.
- Sometimes this pattern suggests a new class.
- In the Sticks game, who creates a Move?
- Refer to the ***creational design patterns***:
  - ***Factory, Builder, Prototype & Singleton***
- In C++ there needs to be a ***destructor*** as well.

High Cohesion

- Cohesion is a measure of the degree to which a class’ responsibilities are ***semantically related***.
- High cohesion promotes:
  - Ease of understanding & maintenance
  - Encapsulation
  - Low coupling
- ***Separate concerns***.
- Counter Example:
  - An entire program could be written with one class and many methods. That’s not OO.

Low Coupling

- Coupling is a measure of how strongly one class has knowledge of, or relies upon other classes.
- Low coupling is encouraged by using ***interfaces***, and the maximum degree of ***encapsulation***.
- Low coupling reduces the complexity of the graph of interacting objects.
- Counter Example: Spaghetti code.

<u>**Law of Demeter**</u>:

- Only talk to your immediate friends:

  `dog.legs.walk()` breaks the law;

  `dog.walk()` does not.

Controller

You often need an object to coordinate other objects.

- Objects have responsibilities which can include controlling and sequencing.
- Commonly used with transactions & program flow.
- Examples:
  - In the Sticks Game, the Referee
  - The ***Transaction Agent*** Design Pattern
  - The ***Mediator*** and ***Façade*** Design Patterns
  - The ***Model - View - Controller*** Design Pattern
  - The class Fractal in the Fractal Applet **`http://www.gui.net/fractal.html`**

Model-View-Controller

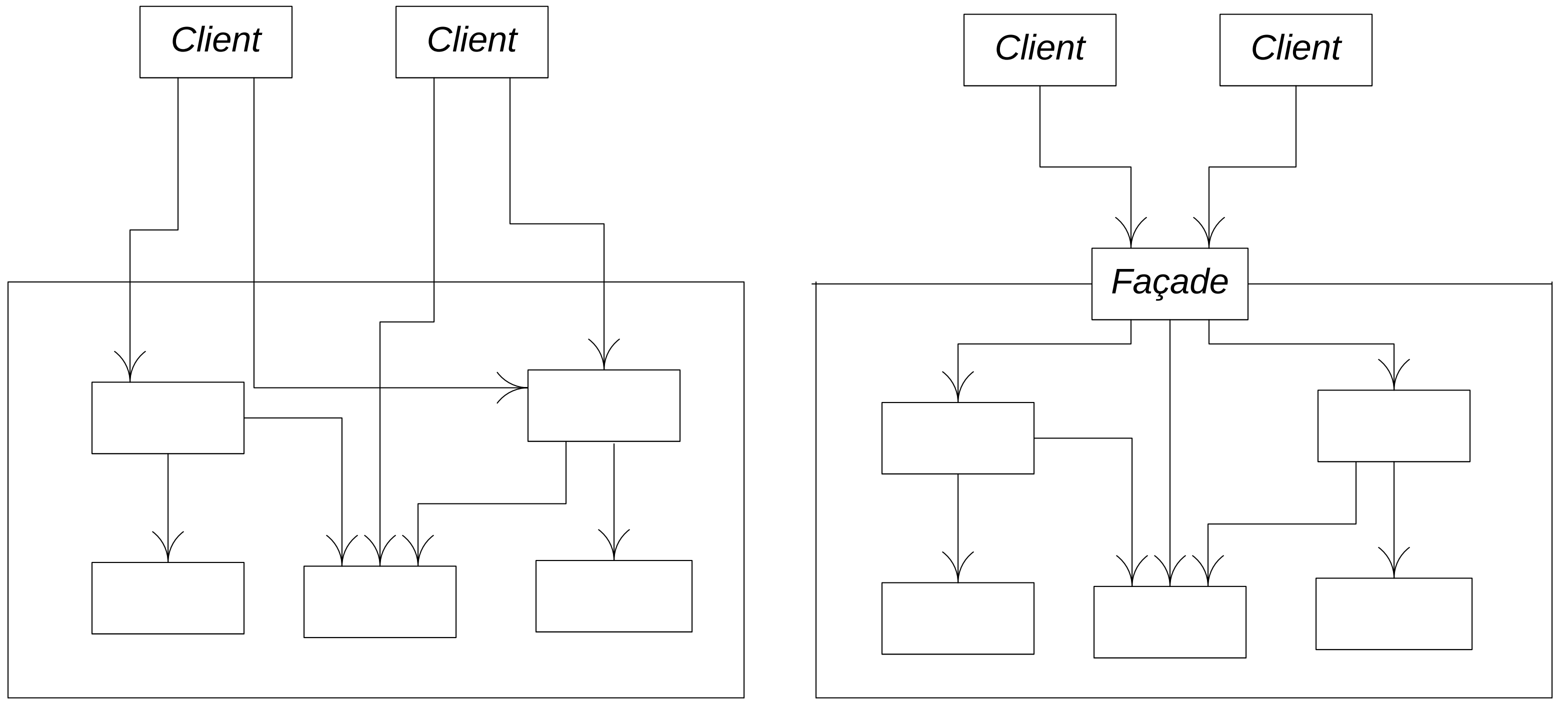
Poorly designed GUIs have classes with a large and incoherent set of responsibilities that include configuring the layout, listening for events, domain logic, application logic, technology specifics, … "spaghetti code" ... We should ***separate the concerns***.

- The **<u>Model</u>** (data plus domain / ***business logic***) knows nothing about the presentation of information to humans.
- The **<u>View</u>** is concerned only with the user interface, handling user events (deciphering the user's ***intent*** from the ***gesture)***, and delegating to the Controller.
- The **<u>Controller</u>** is "the glue" – the ***application logic***. Things like multi-threaded synchronization and transaction control are often done here.

Design Pattern: *Façade*

- **Intent**: Provide a single interface to an architectural layer or component. The Façade class ***controls*** the other classes that make up the sub-system.
- Promotes ***low coupling***; the client knows only about the Façade.
- May compromise ***cohesion***; the Façade class itself can get large.
- Does little more than ***delegate*** to other classes inside the package.
- Often used in distributed applications to increase performance by reducing the number of calls across the network.
- Examples:
  - Modern interface to legacy system
  - Customer Service Representative

Façade Example



- The Façade defines an interface that makes the subsystem easier to use. Clients remain ignorant of the details of the subsystem's components.

Important Guidelines

- Evaluate every decision using principles of good design.
- Implement the basic functionality first, not necessarily the sexiest.
- Don’t expect to always get it right the first time.
- When in doubt, leave it out.
- Design for testing – write unit tests.
- Keep it simple.
- Iterate… refactor.
- Be consistent.
- Make visual models.
- Consider the larger context.

For more on Project Management and Process, refer to section XIX.