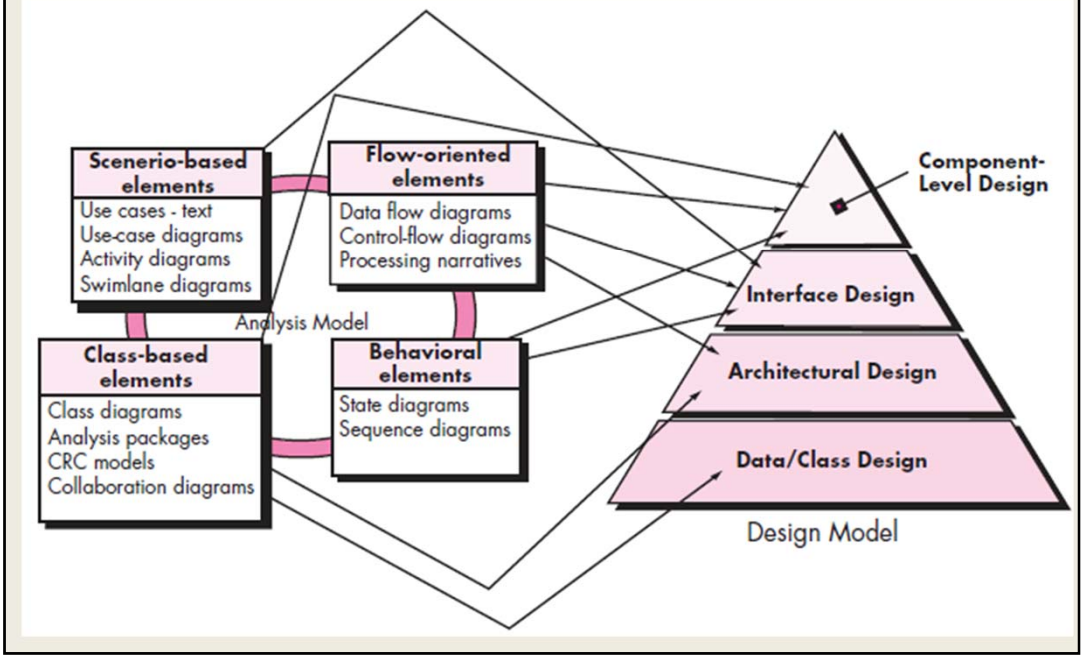
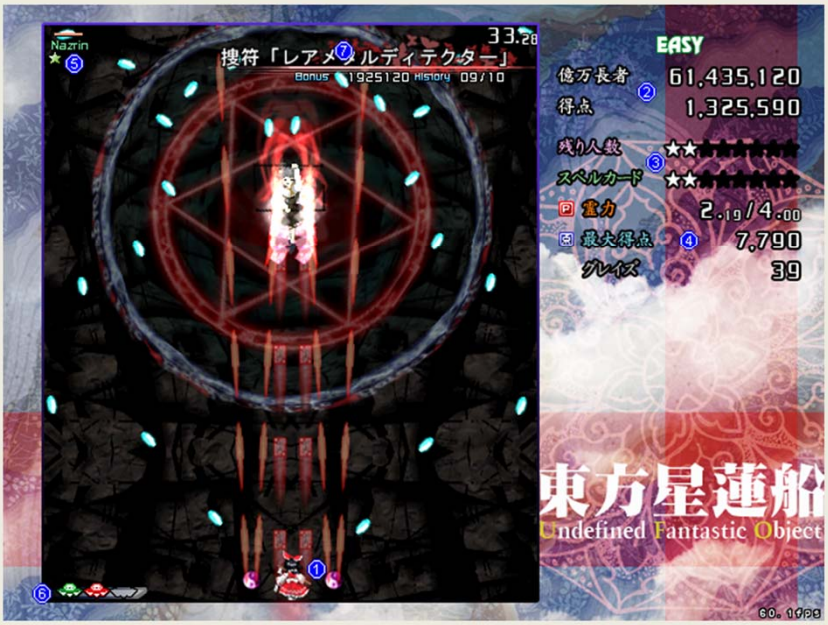
Design modeling review
(a case study)

Translation: analysis to design



We are now going to review the entire process of modeling: including analysis and design modeling

Case study project: a shooter game



Undefined Fantastic Object features three playable characters (Reimu, Marisa and Sanae), each with two weapon types

A single player game.

Playable characters:

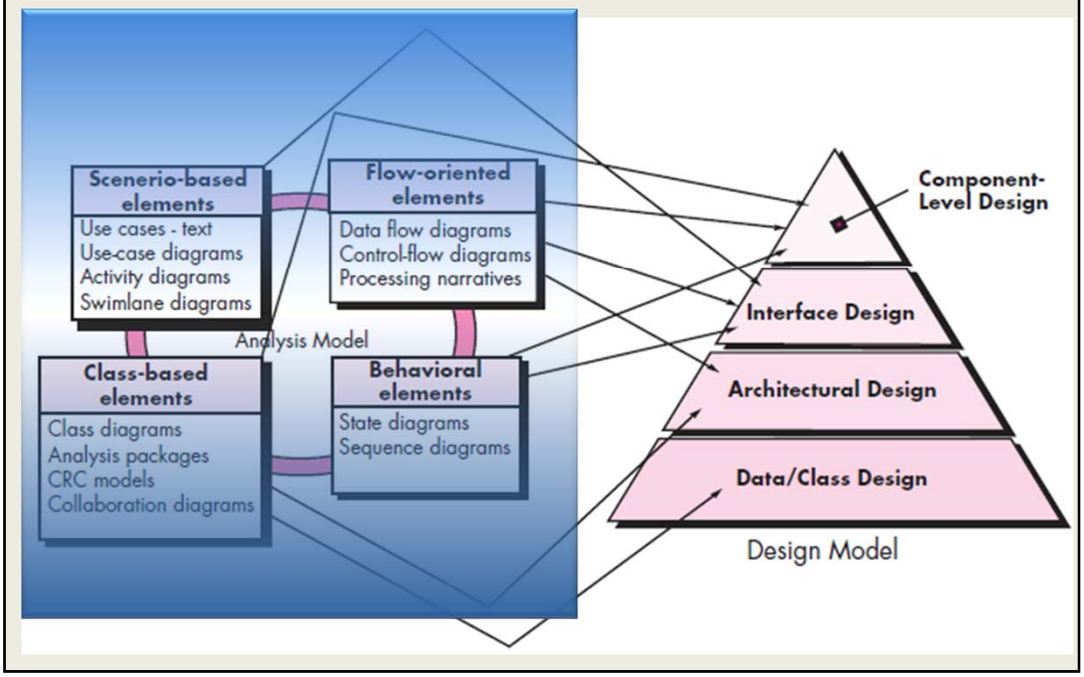
Reimu Hakurei - The miko of the Hakurei Shrine. She sets out to investigate the treasure ship thinking it is yet again the work of yōkai.

Marisa Kirisame - A magician who lives in the Forest of Magic. She seeks the treasure ship simply because it piqued her curiosity.

Sanae Kotiya - The wind priestess of Moriya Shrine who is playable for the first time in *Touhou.* She is still adjusting to life in Gensokyo, and tries to prove herself by setting out to find the treasure ship under the orders of the goddesses she serves.

Enemies: seven bosses, six for stage 1 through 6, and one for extra stage.

Translation: analysis to design



First, requirement analysis/modeling

Preliminary Features

- A “bullet curtain” (danmaku) shooting game.
- You play the game by controlling your character, including moving, shooting, bombing.
- There are multiple difficulty-level, characters, weapons to be chosen.
- There are lots of enemies, both regular ones and bosses, which shoot a large amount of bullets.
- Your character has a limited number of lives. If you lose all of them, game over.
- Killed enemies drop items that can be collected for power, score, and bonus.
- You have to beat 6 stages to win the game. At the end of each stage, you face a boss battle.

These are the high-level features of this game application

There are 4 levels of difficulty: Easy, Normal, Hard, and Lunatic

Features and Use cases

1. Go to a menu (with some options)
2. Select characters
3. Select difficulty levels.
4. Replays
5. Select weapon
6. Blow stuff up.
7. Collect (items) -
8. Rewarding risky behavior
9. Play music.
10. Enemies: regular ones and bosses.
11. Projectiles: different types of bullets.
12. Enemies drop items.
13. Save/Load
14. Setup at the start of the program.
15. Keeping scores.
16. In game upgrades.

Use cases (from player's point of view)
1. Attack/Shoot
2. Move
3. Pause
4. Exit
5. Blow stuff up
6. Select difficulty/character/weapon
7. Die
8. Set up program settings
….

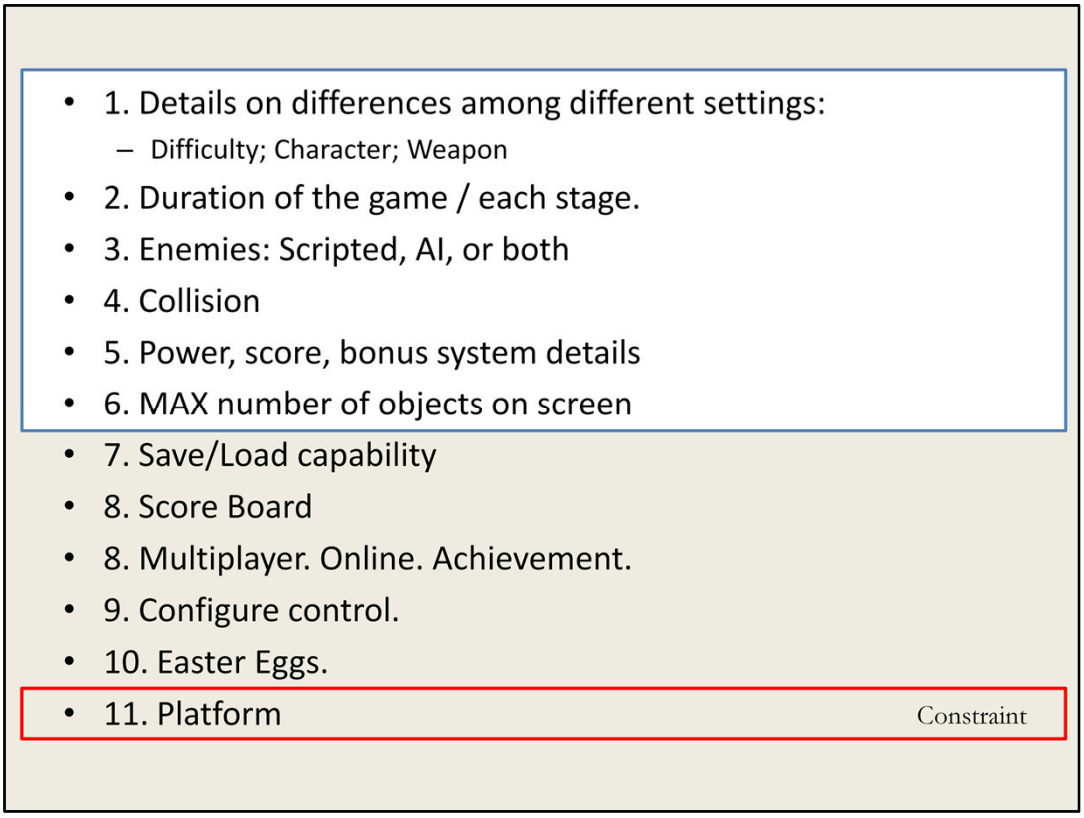
These are the main use cases, corresponding to the main features listed before.

Possible questions to consider

1. Difference between difficulty levels -> differences details
2. How long for the game, with respect to different levels
3. Enemies scripted or AI
4. How to collect the items.
5. Different characters, weapons: details
6. Is there a high score
7. MAX number of enemies
8. Score affects anything?
9. Can you earn more lives
10. Can you save/load the game
11. Can you change the controls? How do you control? Platform?
12. Online functionality? Achievement system?
13. Multiplayer options? Difficulty?
14. Which enemies drop items?
15. Multiple achievements? Bonus levels, extra bosses. Easter eggs.
16. On what platform?

These are the questions that you may have as the software engineering doing the analysis and design;

Could also be questions that you discuss about with your stakeholders



We will mainly look at two parts: features related to gaming rules, and those concerning system constraint

Gaming Rules

- Controlling? Bombing?
- Choose the setting?
- Life system?
- Item system? Power? Score? Bonus?
- Stages? Boss battles?

- Key feature: bullets

In fact, there are several major dimensions to consider regarding the gaming rules

Refined Features

- Gaming Rules
  - Control
  - Win/Lose condition
  - Details w.r.t different settings.
  - Item system. Power system. Score system. Bonus system.
  - Stages. Duration
  - Player: shoot, bomb.
  - Enemy: movement and shoot. Boss: battle.
  - Bullet: type, pattern, etc.
  - Collision.
  - …
- Save/Load/Replay
- Score board
- Practice
- Music room
- Multiplayer/Online function. Achievement system.
- Control Configuration.
- Easter Eggs.

After taking into consideration those dimensions of gaming rules, we may reach a more refined level of features.

The detailed descriptions on functional features are the starting points of developing refined use cases and class-based modeling.

Actual Rules

- Control: high and low speed mode
- Bomb
  - Enter an invincible mode for a limited amount of time. Erase all bullets. Deal large damage to enemies.
- Setting
- Life system
  - Lost a life when hit by any bullet/enemy itself.
- Item system
  - P-Item: increase power; S-Item: increase score; B-Item: bonus towards bomb or life
  - Power system; Score system.
- Stage.
  - Go through a number of enemies/events, and at the end entering a boss battle.

These are more detailed description about the rules, as a result of the refinement

Actual Rules

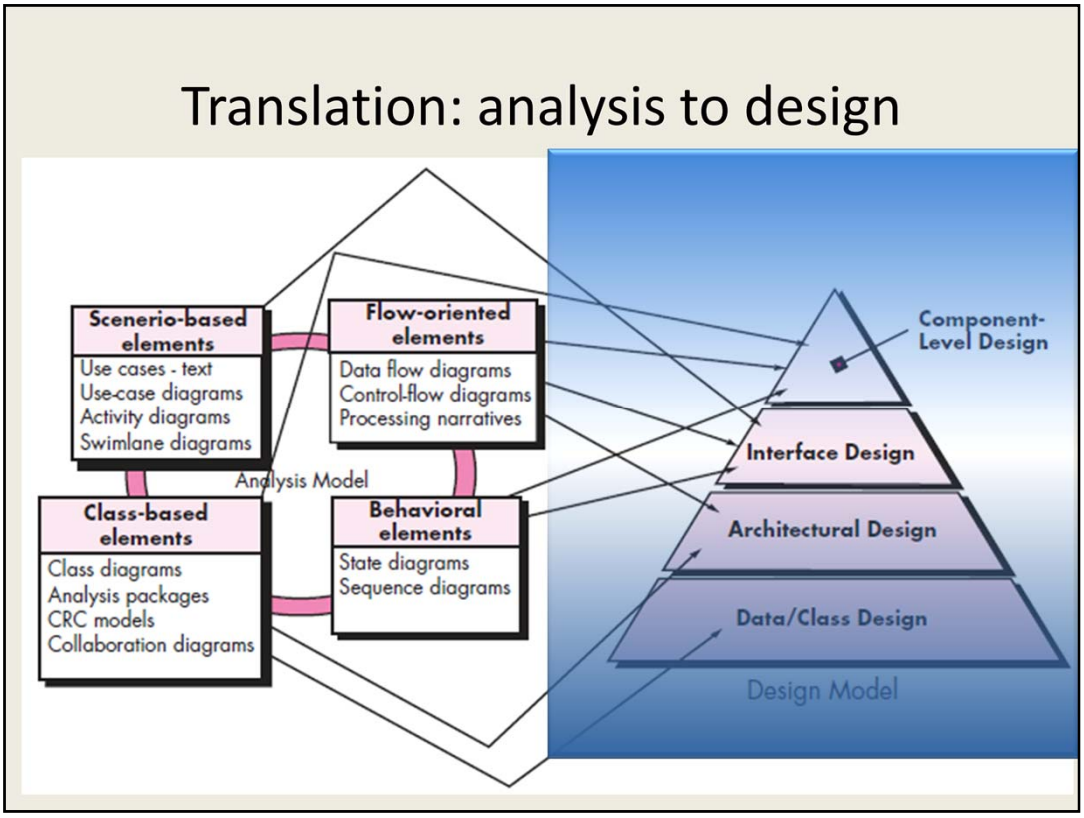
- Boss battle
  - Each boss has several lives. During each one of them, the boss performs a regular attack and then a special attack (called spell card).
- Bullets
  - Types. Speed. Pattern. Size. Trajectory.

Even more details on use cases

Analysis Classes

- Object (Flyer) | Position, Speed, Hitbox | add(); remove(); move();
  - Player | lives, power level, bombs | shoot(); bomb();
  - Enemy | health | reduce_health(); shoot(); drop_item();
    - Boss | lives[], health[], timelimit[] | shoot(); spell_card();
  - Item | type
  - Bullet | …
- Menu | …
- MainScreen |…
- SideBar | …

We already have the use cases, now we should identify analysis classes (i.e., class-based elements in the analysis model)



Now, we move to the design modeling based on the work products from requirements analysis/modeling.

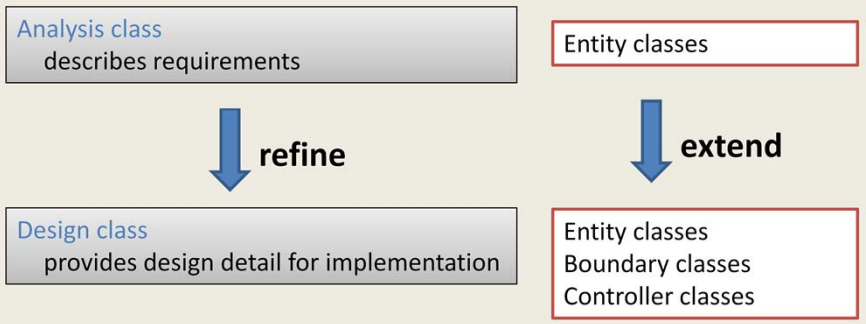
Design Classes

- Analysis classes are refined during design to become entity classes
- Boundary classes are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- Controller classes are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

So, the first step is to do data/class design.

Data/class design

- Design classes



The analysis model defines a set of analysis classes.

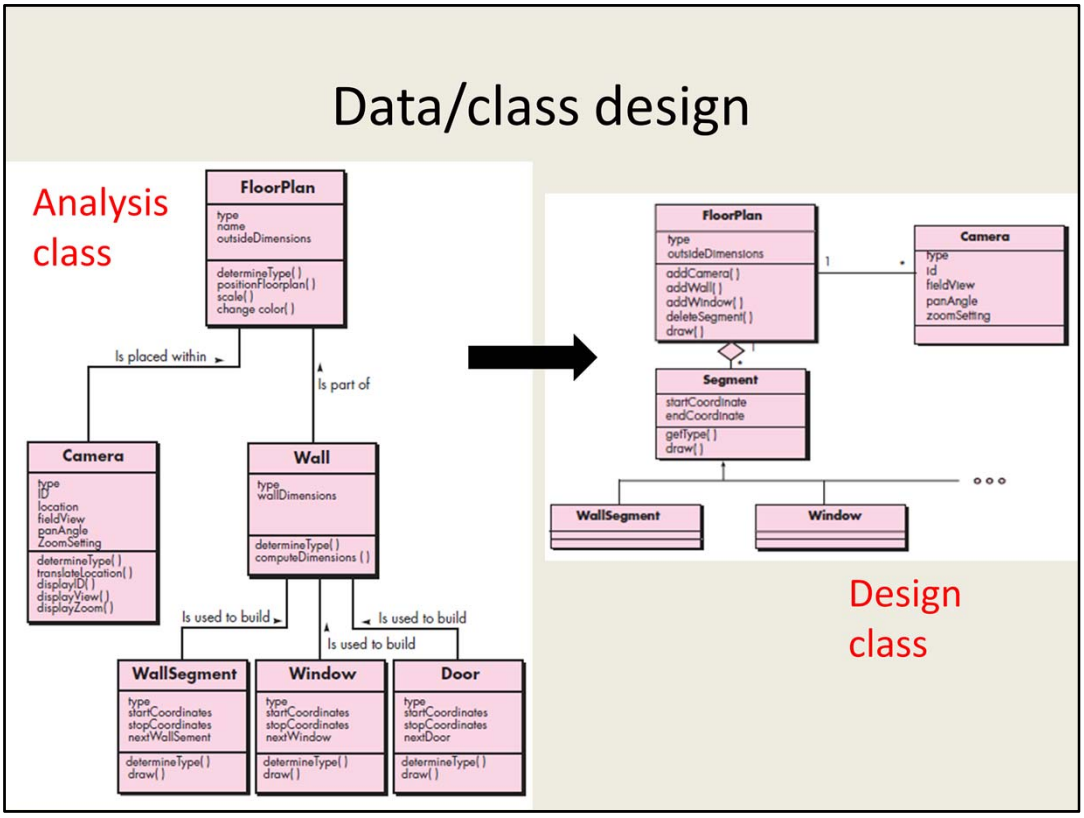
The level of abstraction of an analysis class is relatively high

Design classes refine the analysis classes

Design class provides design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Controller classes are designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, and (4) validation

of data communicated between objects or between the user and the application.



Example (SafeHome) - Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)

The analysis class showed only things in the problem domain, well, actually on the computer screen, that were visible to the end user, right?

Ed: Yep, but for the **FloorPlan** design class, I've got to add some things that are implementation specific. I needed to show that **FloorPlan** is an aggregation of segments—hence the **Segment** class—and that the **Segment** class is composed of lists for wall segments, windows, doors, and so on. The class **Camera** collaborates with **FloorPlan,** and obviously, there can be many cameras in the floor plan.

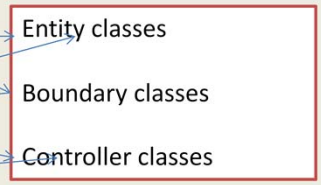
Design Classes

- User interface classes
  - Define all abstractions that are necessary for human-computer interaction.
- Business domain classes
  - Often are refinements of the analysis classes defined earlier.
  - Identify the attributes and services (methods) that are required.
- Process classes
  - Implement lower-level business abstractions
- Persistent classes
  - Represent data stores
- System classes
  - Implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

p230

Data/class design

- Design classes
  - User interface classes
  - Business domain classes
  - Process classes
  - Persistent classes
  - System classes



This is a finer classification of design classes. (the mapping is my personal opinion!)

User interface classes define all abstractions that are necessary for human computer interaction (HCI).

Business domain classes are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

Process classes implement lower-level business abstractions required to fully manage the business domain classes.

Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software.

System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

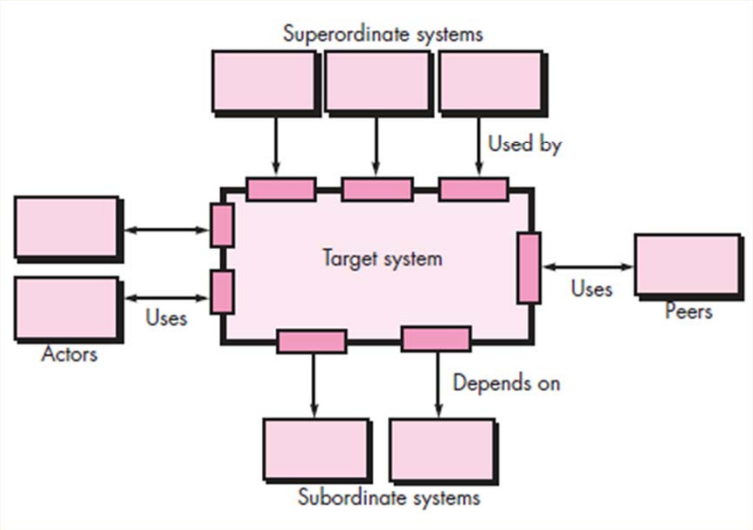
Architectural Design

- The software must be placed into context
  - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
  - An *archetype* is an abstraction (similar to a class) that represents one element of system behavior
    - E.g. Nodes (Detectors and indicators) and controllers, in SafeHome project.
- The designer specifies the structure of the system by defining and refining software components that implement each archetype

20

Architectural design: step 1 (representing system in context)

- Representation: architectural context diagram (ACD)



Systems that interoperate with the *target system*

Superordinate systems—those systems that use the target system as part of some higher-level processing scheme.

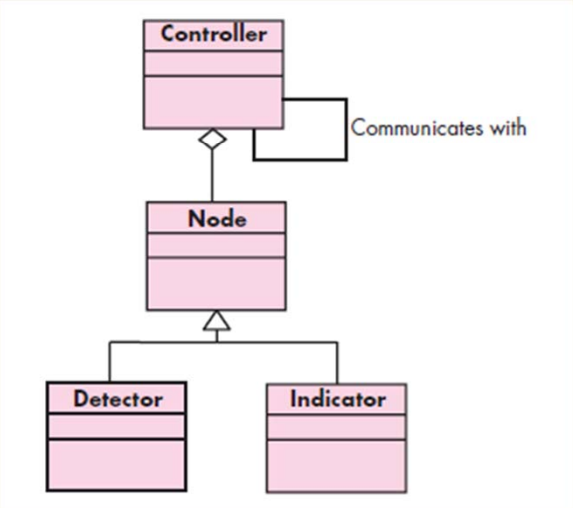
Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

Peer-level systems—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.

Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing

Architectural design: step 2 (defining archetypes)

- Representation: class diagrams



Node. Represents a cohesive collection of input and output elements of the home security function.

For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators

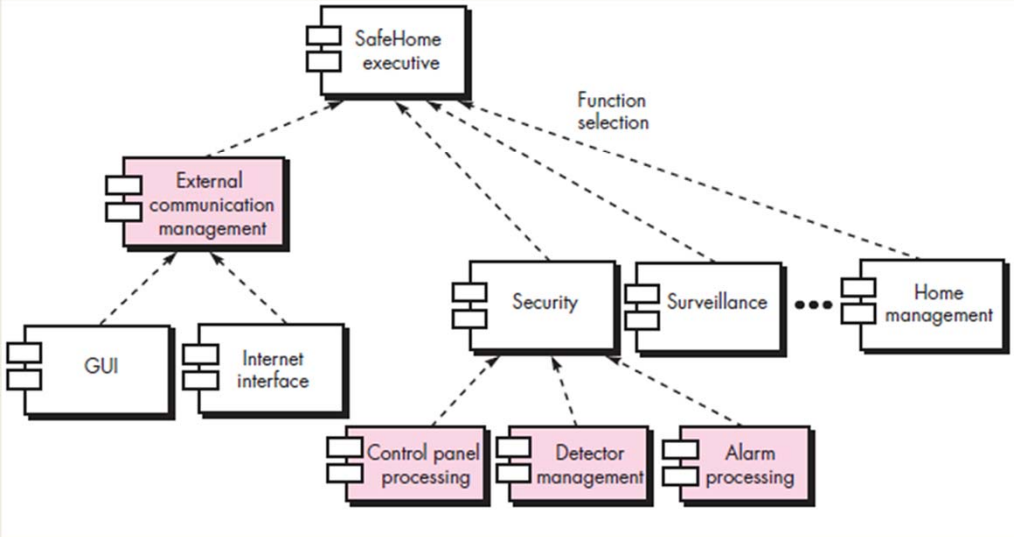
Detector. An abstraction that encompasses all sensing equipment that feeds information into the target system.

Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

Controller. An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

Architectural design: step 3 (refining architecture into components)

- Representation: component diagrams



External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

Control panel processing—manages all control panel functionality.

Detector management—coordinates access to all detectors attached to the system.

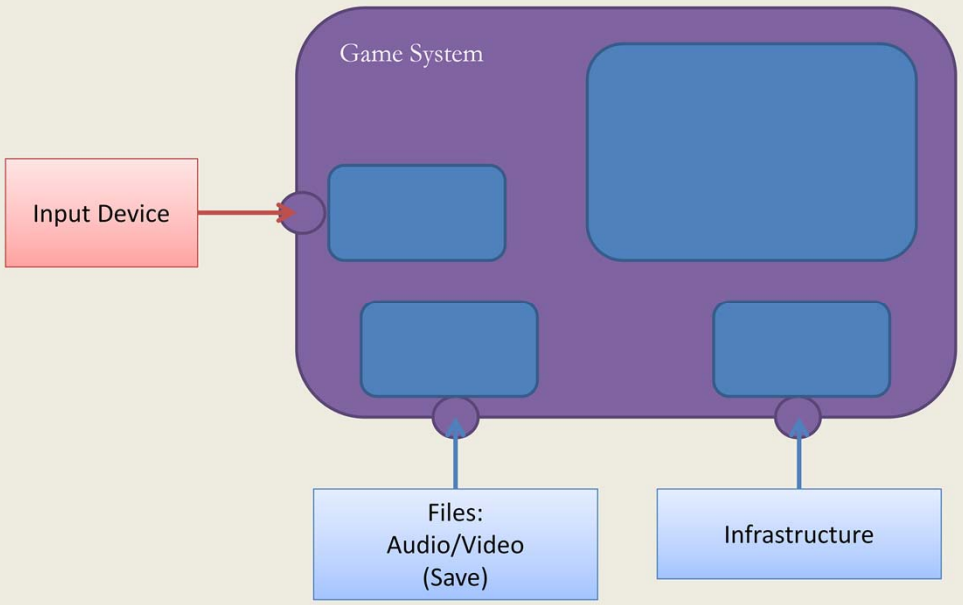
Alarm processing—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture.

Design classes (with appropriate attributes and operations) would be defined for each.

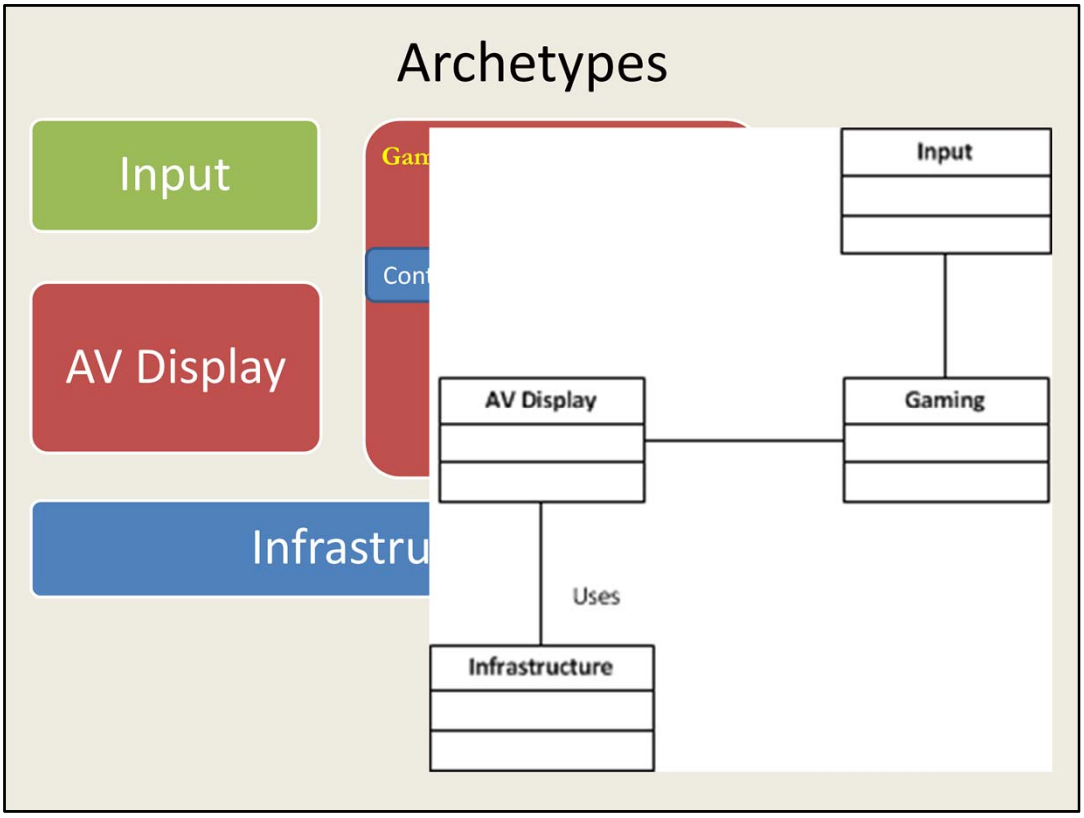
It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design

Architectural Context

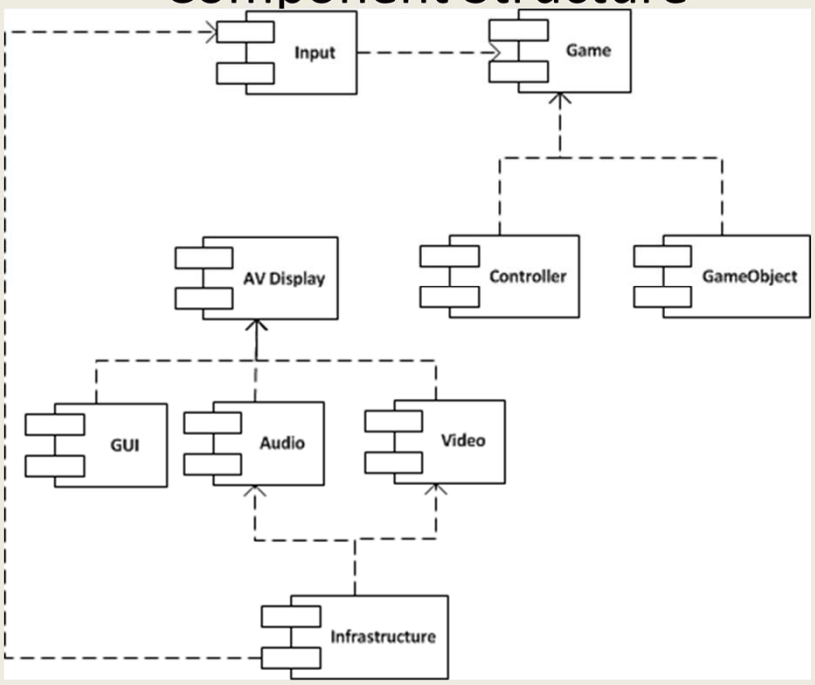


Architectural Context Diagram: Define the relation between a system and its environment

Archetypes


Input
AV Display
Infrastru
Gam
Cont
Input
AV Display
Gaming
Uses
Infrastructure

Component Structure



What is a Component?

- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
  - "… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- OO view: a component contains a set of collaborating classes
- Conventional view:
  - Logic and algorithms
  - Internal data structures that are required to implement the processing logic
  - Interface that enables the component to be invoked and data to be passed to it.

Now, component level design, which we just finished studying

The view of component: OO

- Component: a set of collaborating classes
- Design:
  - Begin with requirements model, elaborate analysis class and infrastructure class
    - Recall: how did we do data/class design?

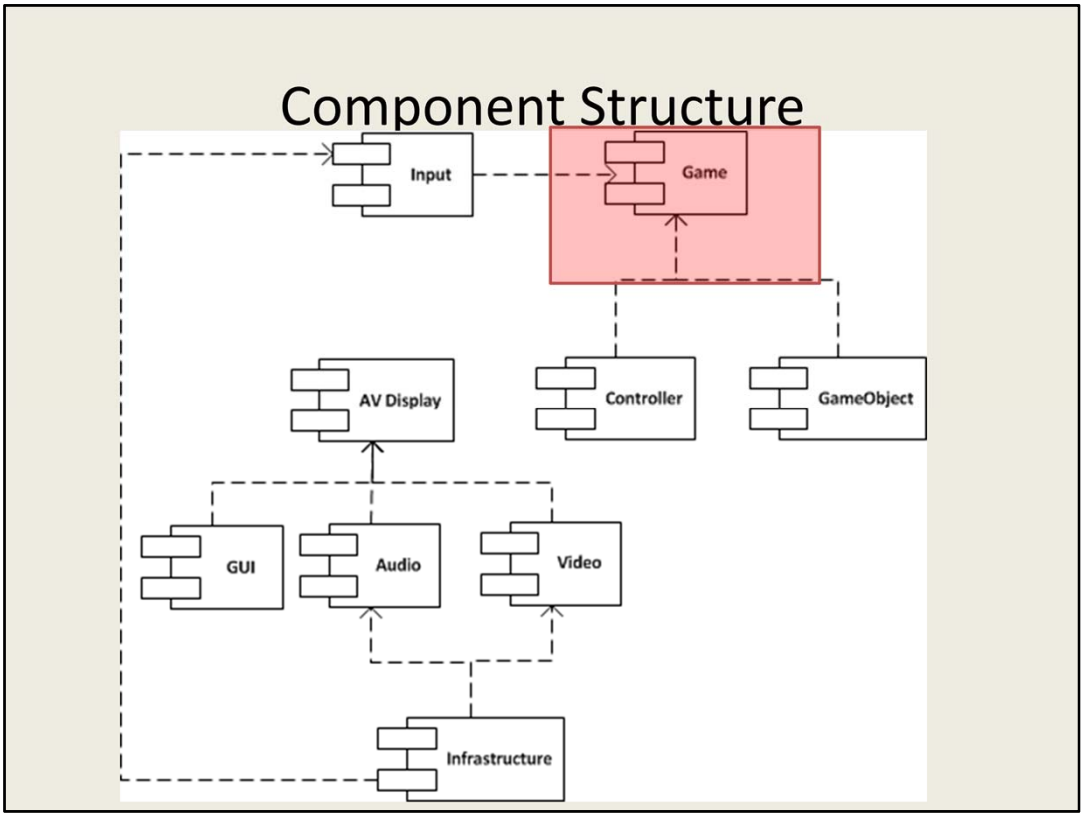
Each **class** within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.

All **interfaces** that enable the classes to communicate and collaborate with other design classes must also be defined.

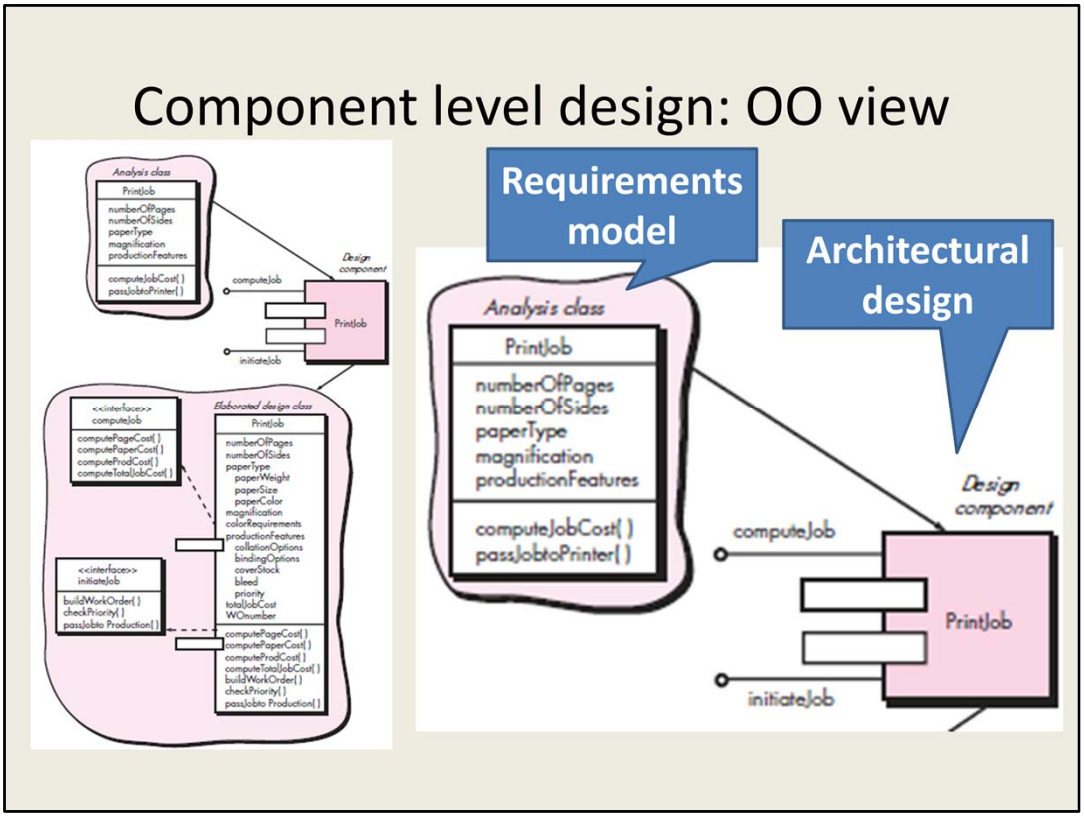
To define a component, we begin with the requirements model and elaborate analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

Select Components

- Analysis classes
- Infrastructure domain
  - Memory management components
  - Communication components
  - Database components
  - Task management components
- Interfaces depicted in the ACD



Pick each component from the architecture design, do the component level design on that component.



During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation shown in the middle right of the figure.

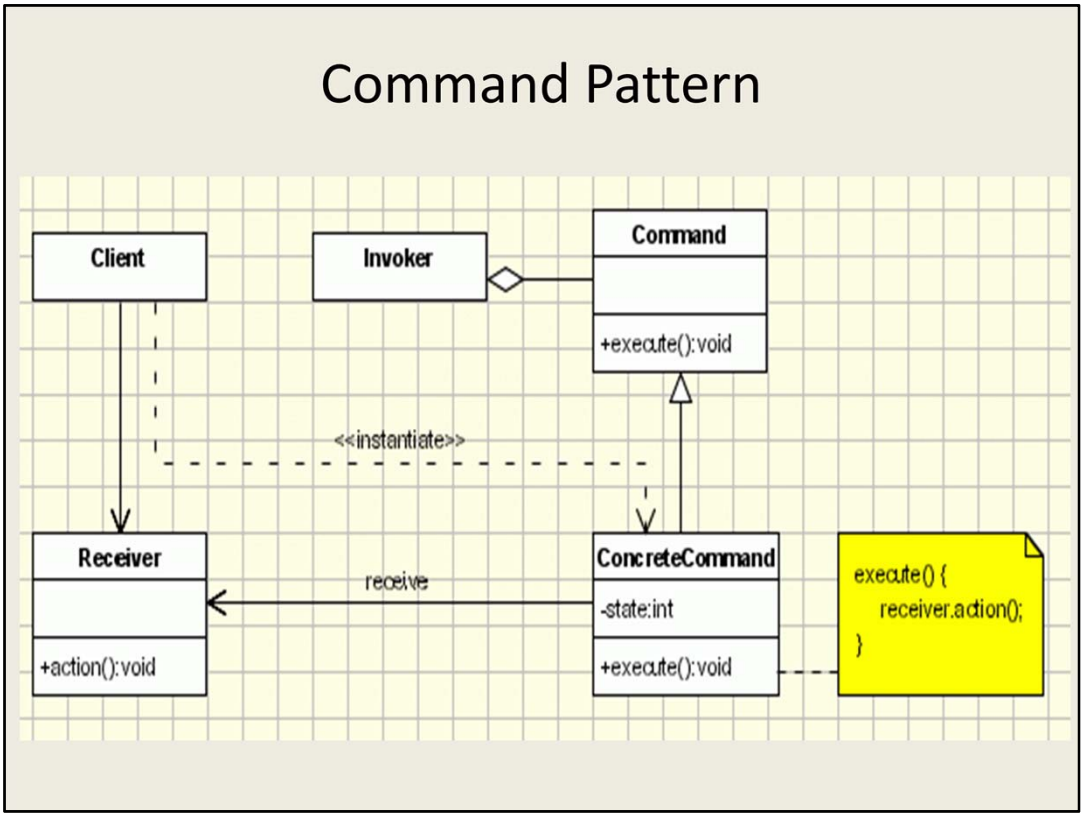
PrintJob has two interfaces, *computeJob,* which provides job costing capability, and *initiateJob,* which passes the job along to the production facility.

Represented using the "***lollipop***" symbols shown to the left of the component box.

Questions (component-level design)

- How do we implement the “behavior” of enemies/bullets?
  - Movement; Action (enemy shooting);
- *Using “scripts” to describe the behavior.*
- How do we control these behaviors? How do the classes interact with each other? (Interface)
  - EnemyManager
  - Enemies
  - BulletManager
  - Bullets
  - Player

For example, when working on the component “Game”, which will implement the major gaming rules, we may get a few questions

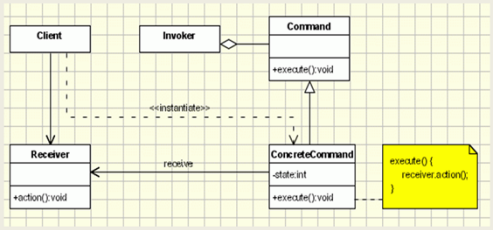


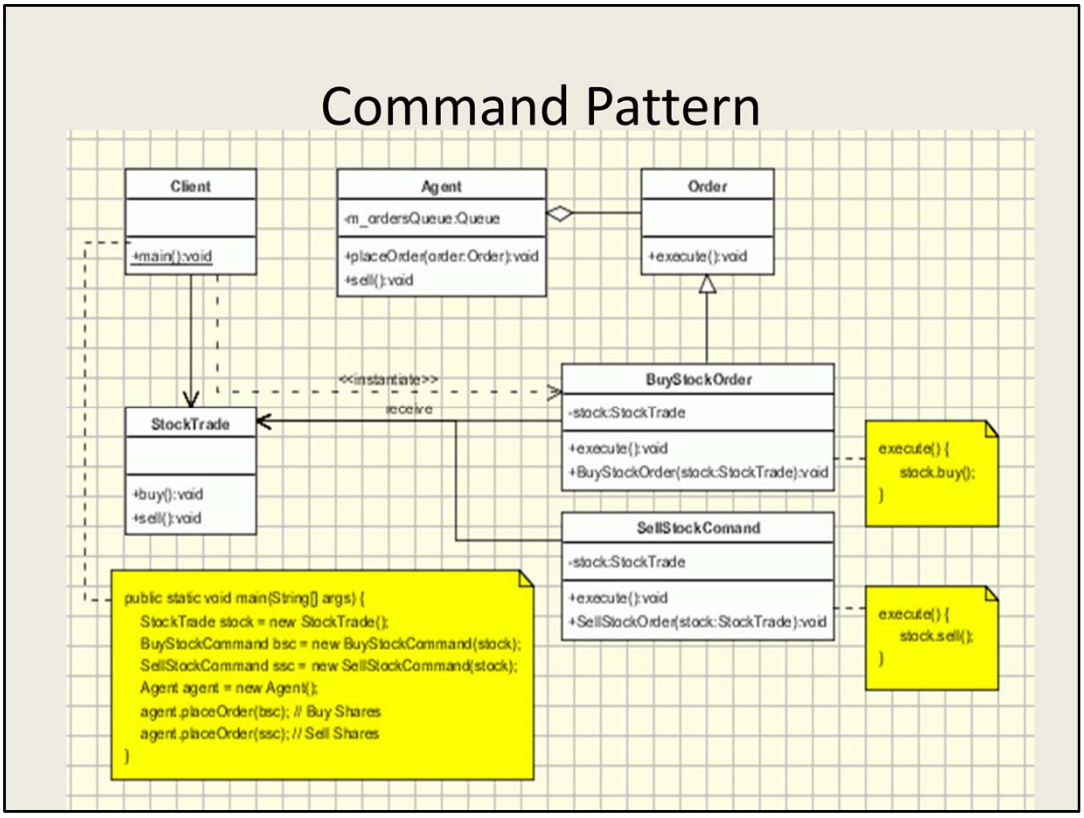
And during the design, we always want to look for appropriate design patterns to adapt.

For this game, we may want to use a pattern called ‘command pattern’

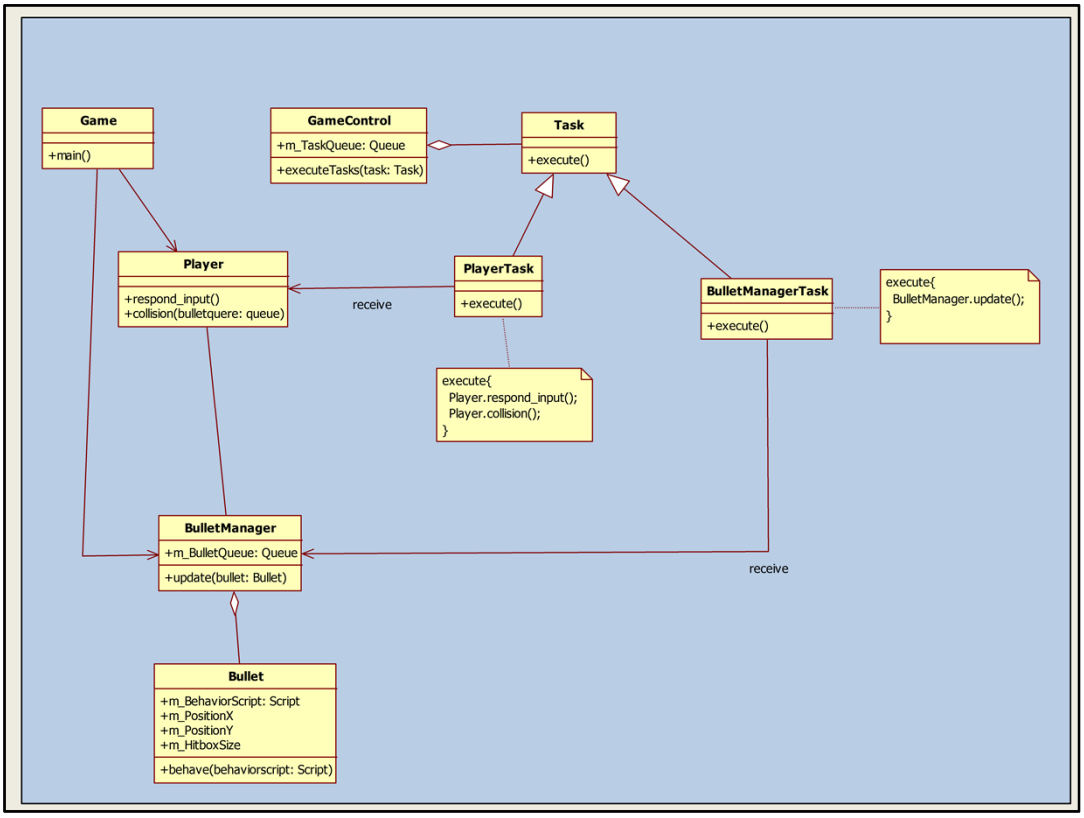
Command Pattern

- Command: declares an interface for executing an operation;
- ConcreteCommand: extends the Command interface, implementing the Execute method by invoking the corresponding operations on Receiver. It defines a link between the Receiver and the action.
- Client: creates a ConcreteCommand object and sets its receiver;
- Invoker: asks the command to carry out the request;
- Receiver: knows how to perform the operations;





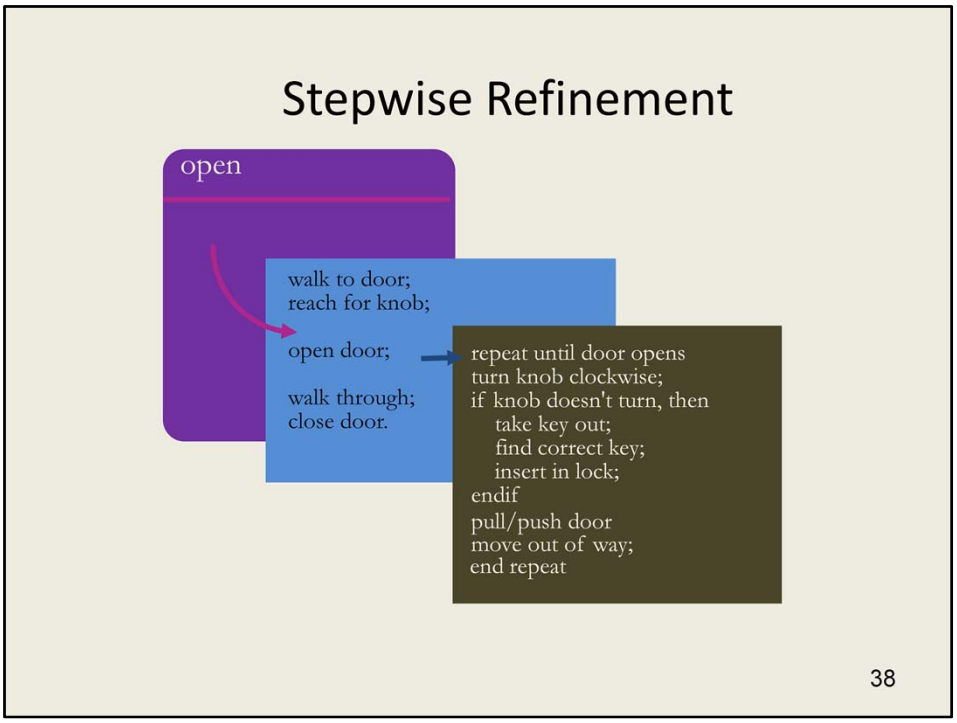
Adapted for the game application.



The component level design of the “Game component”

Design Pattern

- P351
- http://www.dofactory.com/Patterns/Patterns.aspx
- http://www.oodesign.com/



An important design concept that is constantly applied.

Summary

- Requirements and Stakeholders
  - Requirements come from more than just users.
- Prioritize the features
- From requirements analysis to design
- Build up architecture
  - Follow the steps
- Elaborate components
- Look for design patterns
- Address changing requirements