Reading Summaries

Chapter 5 - High-Level Design & Architecture

The second half of Chapter 5 transitions from requirements gathering into high-level design, which is essentially about deciding on the overall structure of a system before you start writing code. Stephens covers a handful of major architectural styles:

Monolithic architecture is where the whole application lives in one big, tightly coupled codebase. It is easy to get started with, but it gets harder to scale or maintain as the project grows.

Component-based architecture (CBA) breaks the system into reusable, self-contained components that talk to each other through defined interfaces. You can swap pieces out, reuse them in other projects, and develop or test them on their own.

Service-oriented architecture (SOA) exposes functionality as services over a network, usually through SOAP or REST. Services are coarse-grained, can be deployed independently, and different clients can consume them.

Client/server architecture splits things between a server (which handles data and business logic) and one or more clients (which handle UI). Your typical two-tier or three-tier web app follows this pattern.

Layered (N-tier) architecture organizes the system into horizontal layers (presentation, business logic, data access) where each layer only talks to the ones directly above and below it.

Pipe-and-filter architecture has data flowing through a sequence of processing stages (filters), each one transforming the data before passing it along. This shows up a lot in data processing and compiler design.

The chapter also gets into database design decisions (flat files, relational, hierarchical, network, object-oriented) and touches on user interface design. The main takeaway is that these architectural choices happen early and are really expensive to undo later.

Chapter 6 - Object-Oriented Design

Chapter 6 is about object-oriented design and how to go from requirements to an actual class hierarchy. It reviews the core OOP concepts: encapsulation (hiding internal state behind a public interface), inheritance (making specialized classes that build on general ones), and polymorphism (treating objects of different classes the same way through a shared interface).

Stephens walks through how to find classes in your requirements by looking for nouns (which might be classes or attributes) and verbs (which might be methods). There is a big emphasis on designing for reuse and following the DRY principle: putting shared behavior in a superclass so subclasses can just inherit it instead of duplicating code everywhere.

UML class diagrams and inheritance hierarchies are introduced as the main tools for communicating OO designs. The chapter also covers state machine diagrams as a way to model

the different states an object or system can be in and the transitions between them. These are particularly useful for things like input parsers, UI flows, and protocol handlers.

Homework Problems

Problem 5.1

Component-based architecture and service-oriented architecture both break a system into discrete, reusable pieces with well-defined interfaces, but the way they do it differs.

With CBA, the unit of reuse is a component, essentially a compiled, in-process module that lives inside the same application. Components communicate through direct method or function calls within the same runtime, so they are relatively tightly coupled since they share a memory space and often a framework. They also get deployed together as one package. You see this a lot in desktop apps, libraries, and plugin systems.

SOA, on the other hand, works with services. These are coarse-grained units that are independently deployable and exposed over a network through things like HTTP/REST, SOAP, or message queues. Because services only interact through published interfaces and can run on completely different machines, they're much more loosely coupled. You can also scale individual services on their own depending on demand, which you can't really do with CBA since you would have to scale the whole app. SOA is the go-to for enterprise systems, microservices, and APIs that get consumed by multiple different clients.

Essentially, CBA is about in-process modularity within a single application, while SOA is about cross-process, network-accessible services that can serve many applications. You could think of SOA as the distributed evolution of CBA.

Problem 5.2

Component-Based Architecture is the best fit here. The tic-tac-toe app is self-contained on one device with no networking involved, so CBA makes the most sense. You can split the app into clean components, a game logic component that manages the board state and checks for wins or draws, an AI component for the computer opponent (maybe a simple minimax or heuristic), a UI component for rendering the board and handling taps, and a score persistence component that reads and writes high scores to local storage. Each one has a single job, can be tested independently, and doesn't need any network overhead.

Layered architecture would also work here. You would have a presentation layer (the game board UI), a business logic layer (game rules, turns, AI decisions), and a data layer (local score storage). In practice, you would probably end up using both, a layered structure where each layer is built out of components.

SOA and client/server are overkill for this. They would introduce network latency, infrastructure complexity, and potential failure modes that just aren't necessary for a single-player game on one device.

Problem 5.4

Once you add two players competing over the Internet, the architectural needs change a lot. Now, you will have to deal with real-time communication, keeping game state in sync, and potentially handling a bunch of concurrent games.

Client/server architecture is the natural choice. Each player runs a client app on their phone that communicates with a central server. The server is the single source of truth for the game state. It validates every move, enforces chess rules, and pushes the updated board to both clients. This is important for preventing cheating, since neither client can just declare a move valid on its own. The server would also handle matchmaking, managing game sessions, and storing things like game history and player ratings.

For the real-time piece, you would want to use WebSockets or something similar rather than REST polling, so each player gets opponent moves instantly without having to constantly ask the server if anything has changed.

If the platform needs to scale up to support lots of users, you could decompose the server side into separate services (SOA/microservices): a game service for managing sessions and validating moves, a matchmaking service for pairing players, a leaderboard or rating service for Elo rankings, and a notification service for pushing move alerts. That way, each piece can be scaled independently, so if the game service needs way more resources than the leaderboard during peak hours.

The big contrast with Problem 5.2 is that going from local-only to networked multiplayer is a fundamental change. You can't just have components sitting on one device anymore; you need a shared server in the middle to keep everything in sync between two different phones.

Problem 5.6

ClassyDraw is a vector drawing app where users create and edit drawings made up of shapes (Lines, Rectangles, Ellipses, Stars, Text, etc.). The right database structure depends on what is being stored and how it will be accessed.

For the actual drawing data**,** a document store or object-oriented database makes the most sense. A drawing is inherently hierarchical. A canvas has layers, layers have shapes, and each shape has its own typed properties (coordinates, colors, stroke weights, text content, etc.). This doesn't map well to a flat relational model, because each shape type has different attributes (a Line has two endpoints, an Ellipse has a center and radii, Text has a string and font) and the inheritance hierarchy of shapes doesn't fit neatly into rows and columns without messy workarounds like

sparse tables or complex joins. Something like JSON or XML files per drawing, or a document database like MongoDB, lets you just serialize the whole object hierarchy directly.

Flat file storage is also totally reasonable for a desktop drawing app. Saving drawings as structured files (SVG, a custom XML format, JSON) means each file essentially is the database record for that drawing. Most real drawing tools work this way: Illustrator uses .ai files, Inkscape uses SVG, and so on.

A relational database would be appropriate for metadata about the drawings, though, not the drawings themselves. Things such as file names, creation dates, modification dates, author info, tags, and folder organization. It is similar to how a photo library stores metadata in a database while the actual images are just files on disk.

For maintenance**,** you would want versioning so users can undo past a single session (like Google Docs version history, where each save creates a new version record), periodic auto-saves and backups to prevent data loss from crashes, cleanup of old undo history versions beyond some threshold so storage doesn't grow forever, and support for standard export formats (SVG, PDF, PNG) so drawings can be shared outside the app.

Problem 5.8

The goal is to recognize floating-point numbers in scientific notation like +37, -12.3e+17, 4.5E-2, or 0.001e10. The grammar allows an optional leading sign, one or more digits, an optional decimal portion, and an optional exponent with E/e, an optional sign, and digits.

I set up states S0 through S7, where S2, S4, and S7 are the accepting (final) states:

S0 (Start): The initial state. On +/− go to S1, on a digit go to S2.

S1: Just saw the optional sign. On a digit go to S2.

S2 (Accept): Accumulating integer digits. This is a valid number on its own. On another digit stay in S2, on '.' go to S3, on E/e go to S5.

S3: Saw a decimal point, need at least one fractional digit. On a digit go to S4.

S4 (Accept): Accumulating fractional digits. Valid decimal number. On another digit stay in S4, on E/e go to S5.

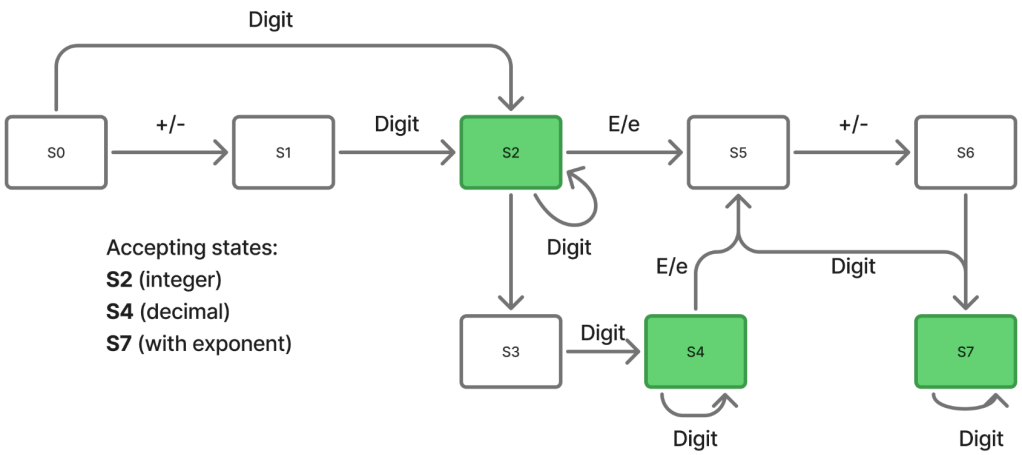
S5: Saw the exponent symbol. On +/− go to S6, on a digit go to S7.

S6: Saw the exponent sign. On a digit go to S7.

S7 (Accept): Accumulating exponent digits. Valid number with exponent. On another digit stay in S7.

Any input not listed for a given state leads to a reject/error state.

State machine diagram:



Tracing through the examples to verify:

+37: S0 → S1 → S2 (accept)

-12.3e+17: S0 → S1 → S2 → S3 → S4 → S5 → S6 → S7 (accept)

4E2: S0 → S2 → S5 → S7 (accept)

12.: S0 → S2 → S3 (reject – decimal point with no digit after it)

Problem 6.1

Considering the ClassyDraw classes: Line, Rectangle, Ellipse, Star, and Text.

Part (a) - Shared properties

All of these classes share: position (x, y coordinates of the anchor point), stroke/foreground color, stroke width, visibility (whether the shape is shown or hidden), z-order (stacking position relative to other shapes), and selection state (whether the shape is currently selected).

Part (b) - Properties NOT shared

Text has font family, font size, font style (bold/italic), the actual text string, and text alignment, and none of which apply to the geometric shapes. Line has its own start point (x1, y1) and end point (x2, y2), and Lines don't have a fill. Rectangle has width, height, and corner radius for rounded corners, plus a fill color. Ellipse has radiusX and radiusY (or center point plus semi-axes), plus a fill color. Star has number of points, outer radius, and inner radius, plus a fill color.

Part (c) - Properties shared by some but not all

Fill color is shared by Rectangle, Ellipse, Star, and arguably Text (for background), but not by Line since lines are just strokes. Width and height as bounding dimensions apply to Rectangle, Ellipse, Star, and Text, but a Line is defined by its endpoints rather than width/height. Rotation angle applies to Rectangle, Ellipse, Star, and Text (all can rotate around their center), but a Line's angle is just inherent in where its endpoints are.

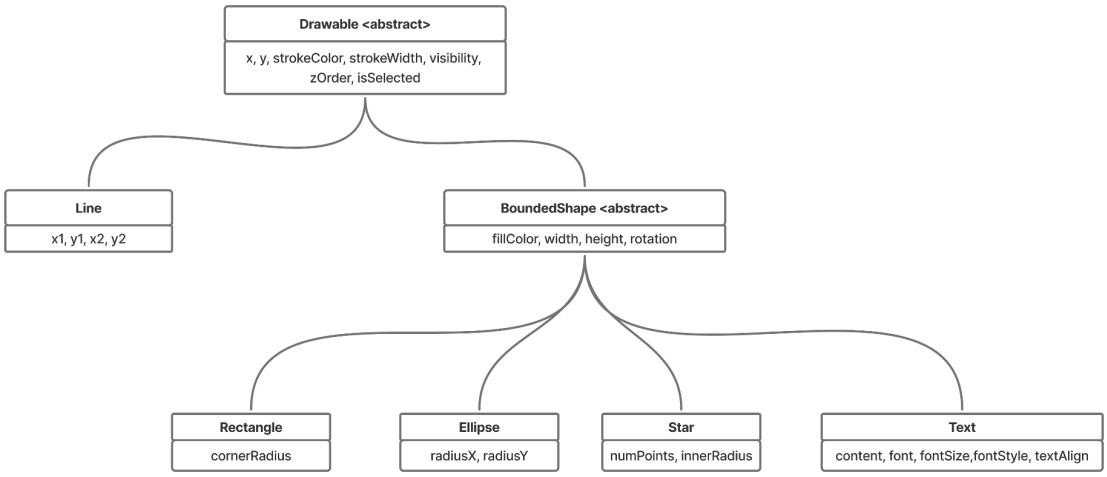
Part (d) - Where to implement them

The properties that every shape shares (position, stroke color, stroke width, visibility, z-order, selection state) should go in a common superclass so all shapes inherit them. This follows the DRY principle. Properties shared by most shapes but not all (fill color, width/height, rotation) should go in an intermediate superclass for filled, bounded shapes that Line doesn't extend. Properties that are unique to one class (text content, font stuff, number of star points) should stay in their specific subclass.

Problem 6.2

Properties listed next to each class are the ones new at that level

Inheritance hierarchy:



Drawable is abstract and sits at the top. It has everything that's universal to all drawable objects. You never make a plain Drawable instance.

Line extends Drawable directly and skips BoundedShape. Lines don't have a fill and don't really have a width/height in the same way since they're just defined by two endpoints. So it wouldn't make sense for them to inherit fill color or bounding box dimensions.

BoundedShape is another abstract class that extends Drawable. It’s for shapes that have a fill color, a bounding box (width and height), and can be rotated. Rectangle, Ellipse, Star, and Text all fit this description.

Then Rectangle, Ellipse, Star, and Text each extend BoundedShape and just add their own unique properties: corner radius for Rectangle, semi-axes for Ellipse, point count and inner radius for Star, and all the text-specific stuff (content, font, size, style, alignment)