Product Design

JD Kilgallin

CPSC:480

09/26/22

Pressman Ch 9-11, Appendix 1

There are two ways of constructing a software design – make it so simple that there are obviously no deficiencies, [or] make it so complicated that there are no obvious deficiencies. -C.A.R. Hoare

Notes

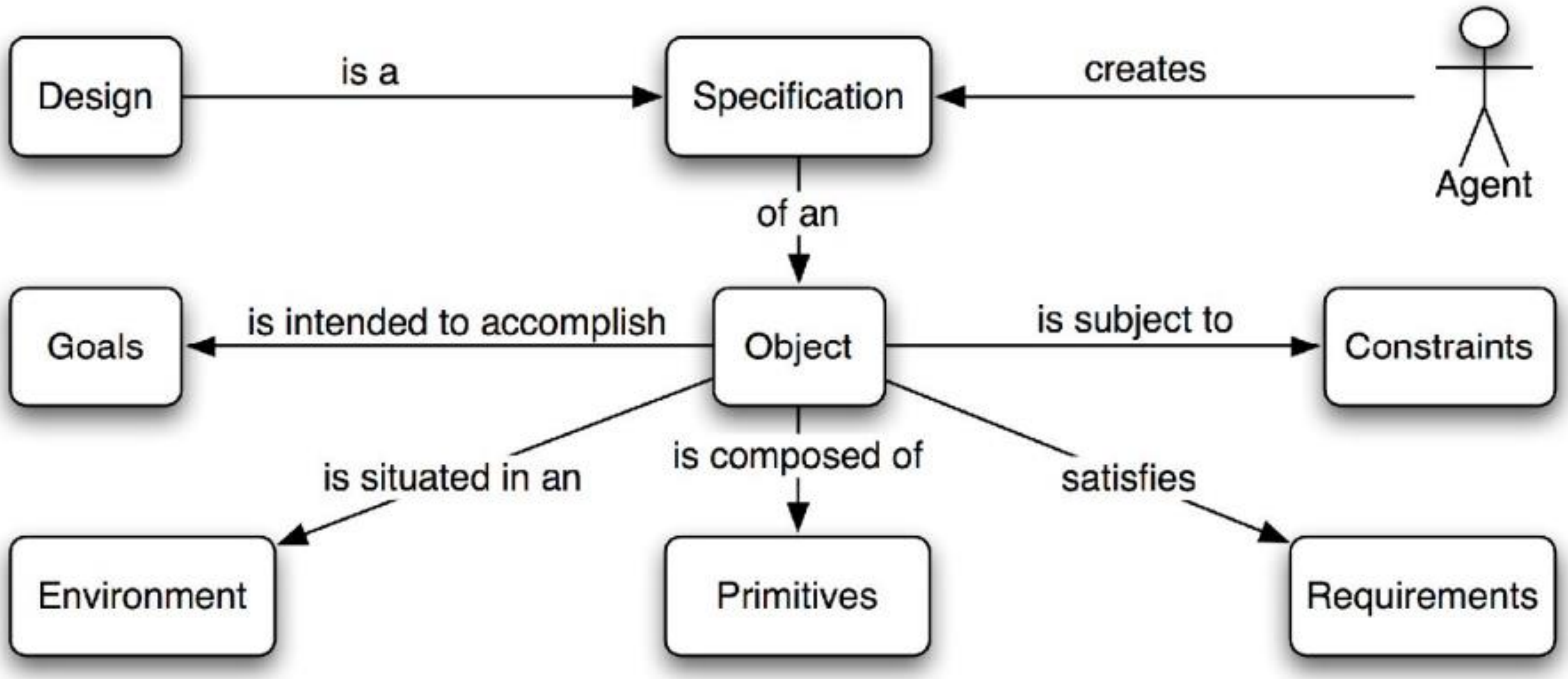
- Resume feedback is out. Engineering career fair is tomorrow.
  - I will be at the Keyfactor booth part of the day.
  - I recommend submitting a resume to Keyfactor through me instead of the career fair for more likely consideration.
- Exercise 2 grades are up. I will have a sample Pokedex solution on GitHub soon, constructed partly from student submissions.
- Participation bonuses as of today are up. Will update again in October.
- Project 2 proposals are due tonight at 11:59 PM.
- Quiz Wednesday on lectures 7-10. Same format as quizzes 2 and 3. Exercise 3 in class after the quiz.
- Midterm next Wednesday in class. Review and office hours Monday. Will be a mix of short-answer and modeling. Will reuse quiz and textbook problems. Exam will be long, but curved. **1 page of notes (2-sided) allowed**

Learning objectives

- Design concepts
- Software architecture
- Design patterns
- Effective design principles
- UML diagrams

What is design?

- Deliberate purposeful planning
- A plan (with more or less detail) for the structure and functions of an artifact, building or system.
- The arrangement of elements or details in a product or work of art.
- (noun) a specification of an object, manifested by an agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to constraints;
- (verb, transitive) to create a design in an environment


Design
is a
Specification
creates
Agent
of an
Goals
is intended to accomplish
Object
is subject to
Constraints
is situated in an
is composed of
satisfies
Environment
Primitives
Requirements

What is software design?

- The process of envisioning and defining software solutions to one or more sets of problems.
- The process by which an agent creates a specification of a software artifact intended to accomplish goals, using a set of primitive components and subject to constraints.
- The way to translate stakeholders' requirements into a specification of a finished software product or system.
- A plan for the software *product,* rather than the development *project.*
- (noun) A design in which the specification is for a software system.
- (verb) The process of creating a software design.

Why is software *designed?*

- A software design can be assessed for quality and suitability before investment in construction of the software as designed.
- Allows final product to be assessed for completeness and quality.
- A software design defines an end goal that will determine when software construction is complete, and allow measurement and forecast of progress toward completion.
- A software design can provide a common understanding among non-technical stakeholders and engineers who will implement the design.
- Allows for changes in requirements to be translated into changes in software implementation.

When is software designed?

- Before planning: a rough architecture must be established as a reference in order to develop a reasonable plan for other activities.
- Between planning and construction: a project plan must allocate time for design of individual components and their relationships and interfaces in order to begin meaningful construction and avoid cost of design changes after development has begun.
- During construction: Specific design of individual features, classes, interfaces, and methods can take place with some independence; construction does not need to wait for full design of all elements.
- During validation: Unavoidably, some undiscovered or changed requirement may necessitate late-stage redesign of some elements.

How is software designed?

- Incrementally decreasing levels of abstraction.
- Requirements translate into a list of use cases, and objects to be modeled become evident. Requirements models translate to design.
- Five major types of design:
  - Architecture design describes core features and basic relationships needed to model use cases.
  - Component-level design gives more granular specifications of sub-systems and precise relationships within and between them, listing individual object types, their attributes and possible interactions.
  - Interface design describes interactions with users and other systems.
  - Deployment design describes where and how software runs.
  - Data design describes information representation.

Planning vs Design

- Requirements models and artifacts from project planning are user-oriented; they describe what users need to be able to do with the software to accomplish their goals.
- Product design is software-oriented; designs specify how the system works, which may include implementation-specific aspects like:
  - How individual UI elements behave
  - What security model is in place to prevent unauthorized use
  - How data is serialized in web requests/responses
  - What information is recorded in application logs
- Design may specify how the system can be maintained, supported, and extended to meet future use cases that aren't specified.
  - e.g. Keyfactor routinely has to manage certificates/keys on new platforms, and the product is designed with extensibility points to facilitate connecting these platforms without architectural changes by writing a new implementation of an interface.

Use case

- A sequence of actions to accomplish a goal within a system, in a detailed, software-oriented manner (vs a user story being brief and user-centered).
- A technique (and the resulting artifact) for capturing and modeling one requirement of a software system.
- Likely specified in a template for the definition of a use case within the team's project management software. May include fields such as:
  - Title and description ("Request a digital certificate to be issued")
  - Steps taken in the primary scenario, and possibly secondary scenarios
  - Success criteria ("An authorized user is able to input certificate request information and receive a certificate with the specified content")
  - Guarantees ("All certificate requests are logged", "Certificate is permanently stored within the platform", "Invalid input is rejected before submission")

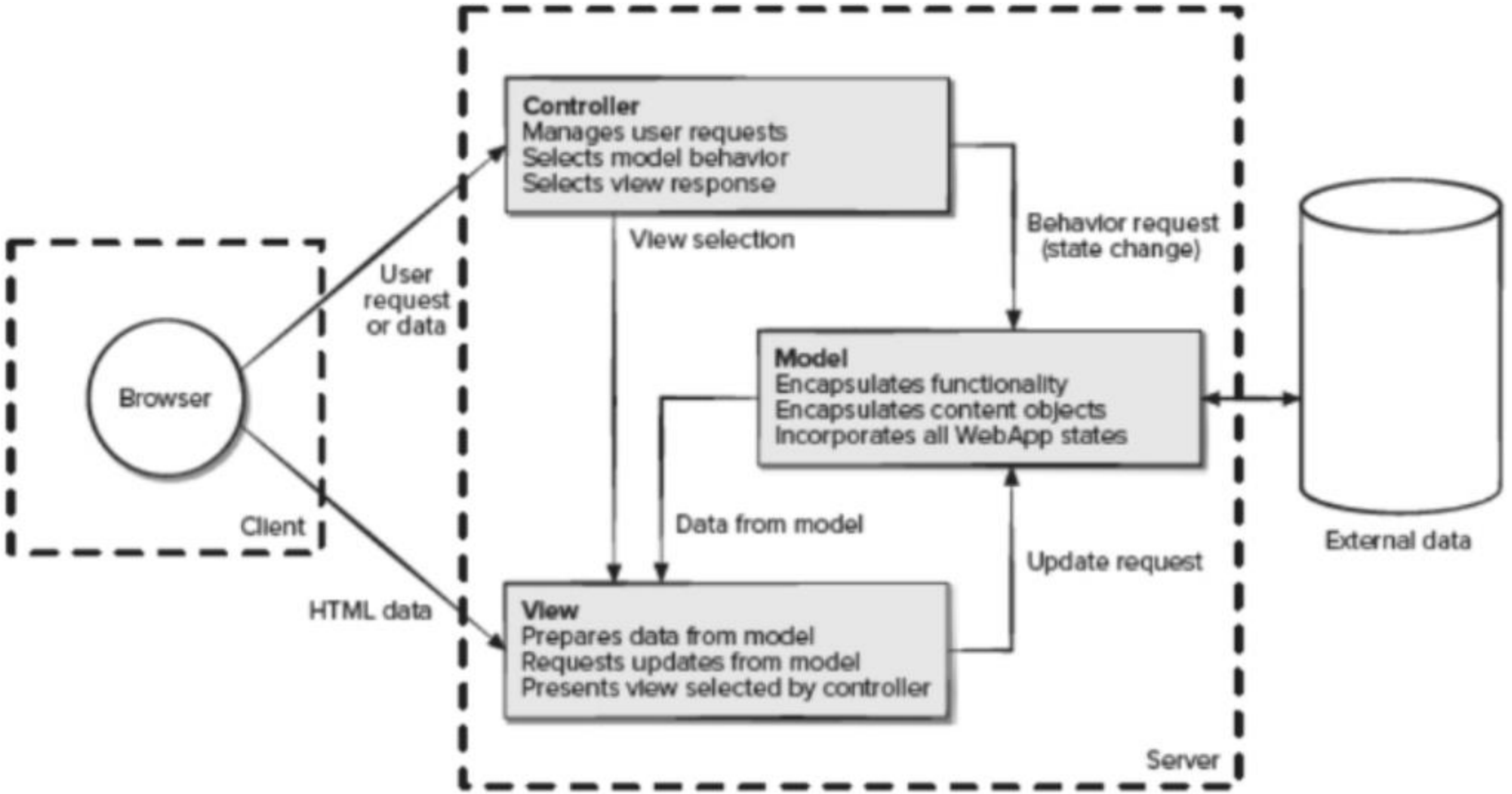
Software architecture

- The relationship between major structural elements of the software, along with the style and patterns that can be used to achieve the requirements defined for the system subject to applicable constraints.
- The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.
- A structured framework used to conceptualize software elements, relationships and properties.
- A process whereby software is decomposed into distinct but interrelated modular components in order to limit complexity and facilitate orderly implementation, even as requirements change.

Architecture patterns

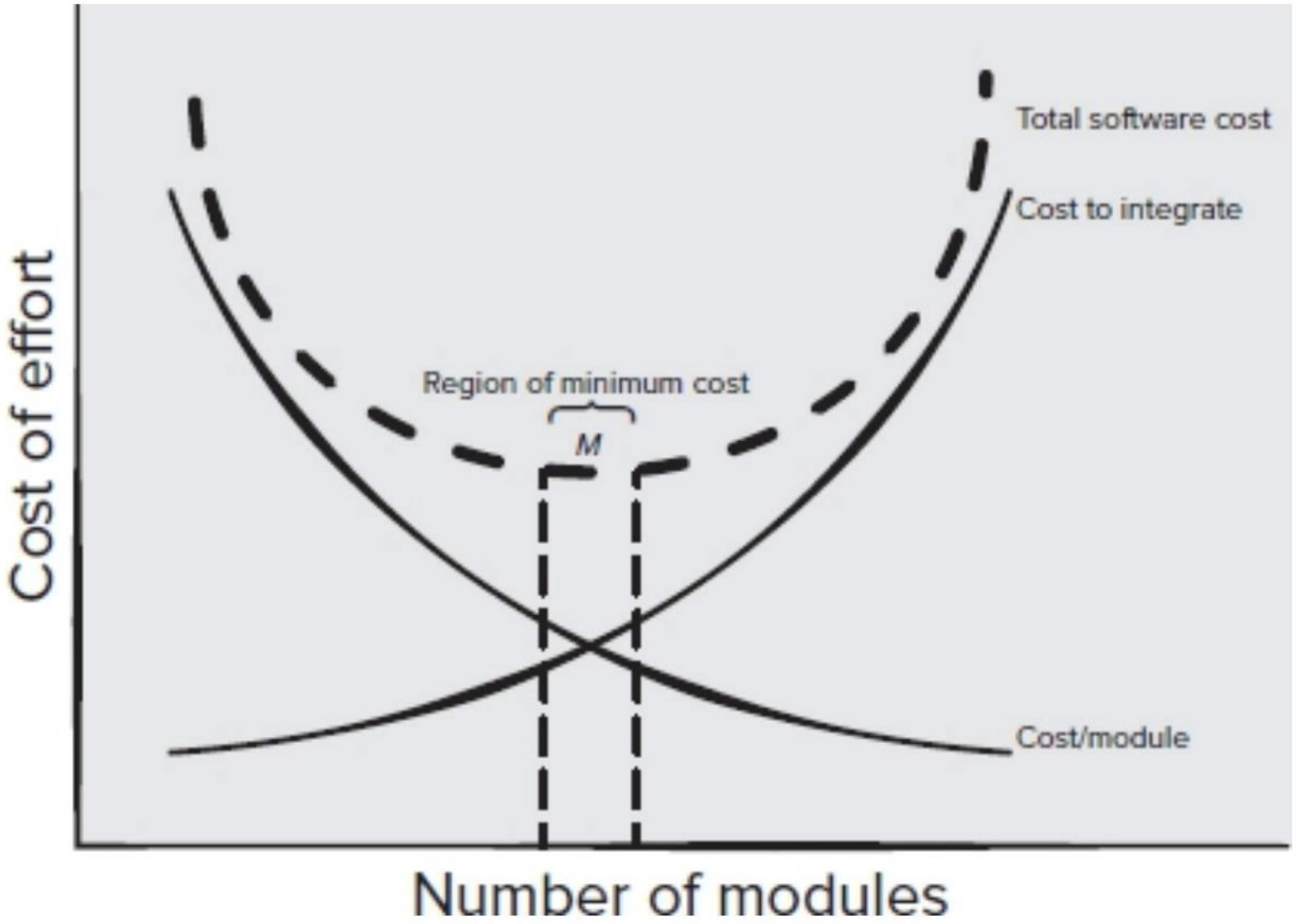
- Reusable solutions to a class of problem that have been proven effective
- Examples:
  - Master-slave: Master component distributes tasks to worker nodes.
  - Pipe-filter: Data streams moves through a series of discrete stages with an operation on each, like a conveyor belt. Compilers, CI/CD pipelines.
  - Broker, publisher-subscriber, or producer-consumer: Services produce data and other services subscribe to receive data produced by those services, without any service needing to know about the existence of other services. Highly scalable.
  - Adapter – Facilitating communication between modules by wrapping one interface in another.
  - Model-View-Controller (MVC) - Decomposition of a program into data management, application logic, and user interface/presentation

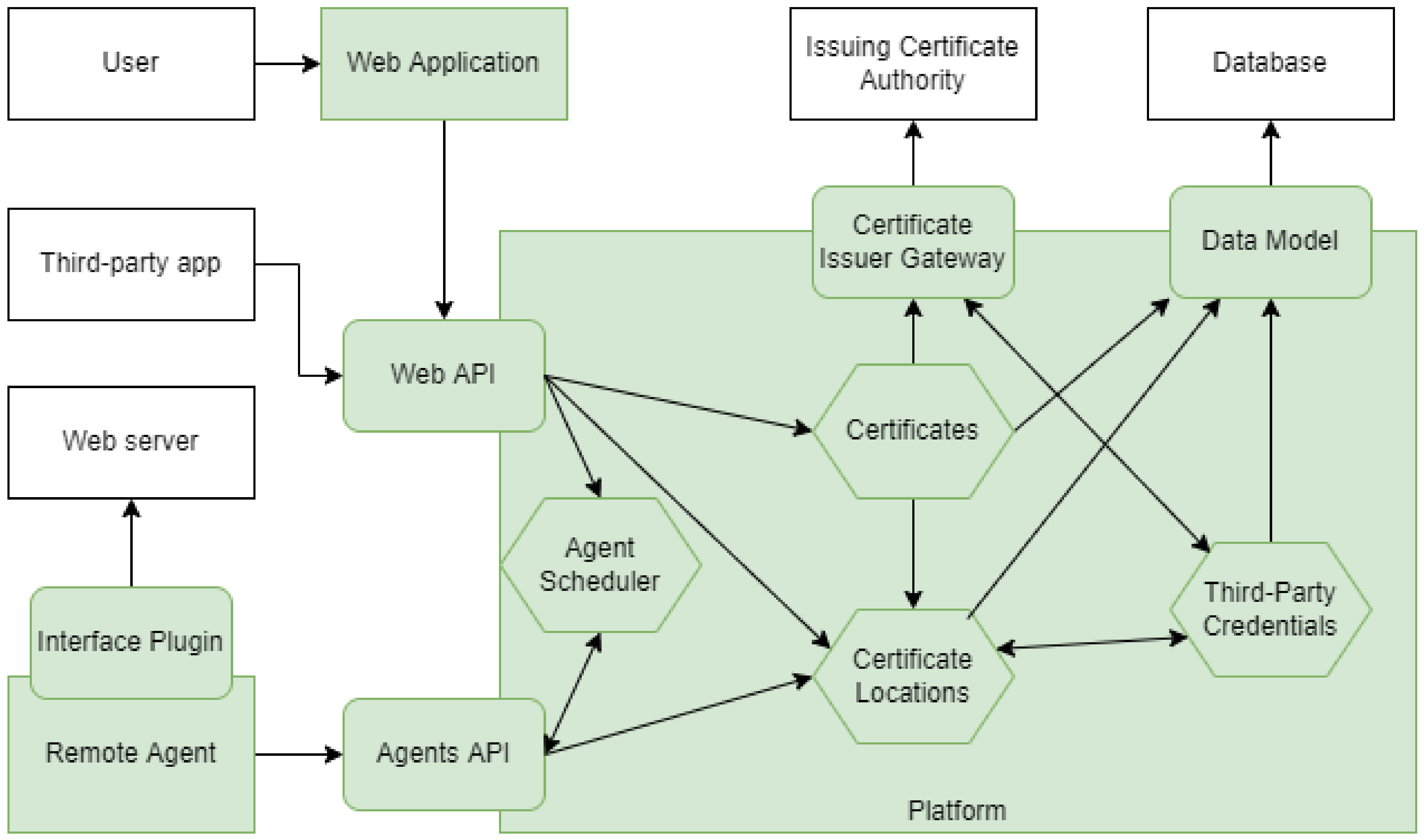
Architecture Pattern: Model-View-Controller (MVC)



Separation of concerns

- A *concern* is a feature or behavior of the software.
- Decomposition of a program into independent components allows a complex system to be broken down into manageable pieces. "The [complexity of the] whole is greater than the sum of its parts" implies "each part is less [complex] than its share of the whole".
- Modularity – Separate, named components that are integrated to comprise the whole program.
- Functional independence – Modules created in a way that each serves a single concern and communication between modules is simplified.
    - Cohesion – Degree to which all parts of a module serve one concern. Maximize this.
    - Coupling – Degree to which a module depends on others for its concern. Minimize.
- Encapsulation – The ability for modules to work together without needing any information from other modules outside of their defined interfaces.


Total software cost
Cost to integrate
Region of minimum cost
M
Cost/module
Cost of effort
Number of modules

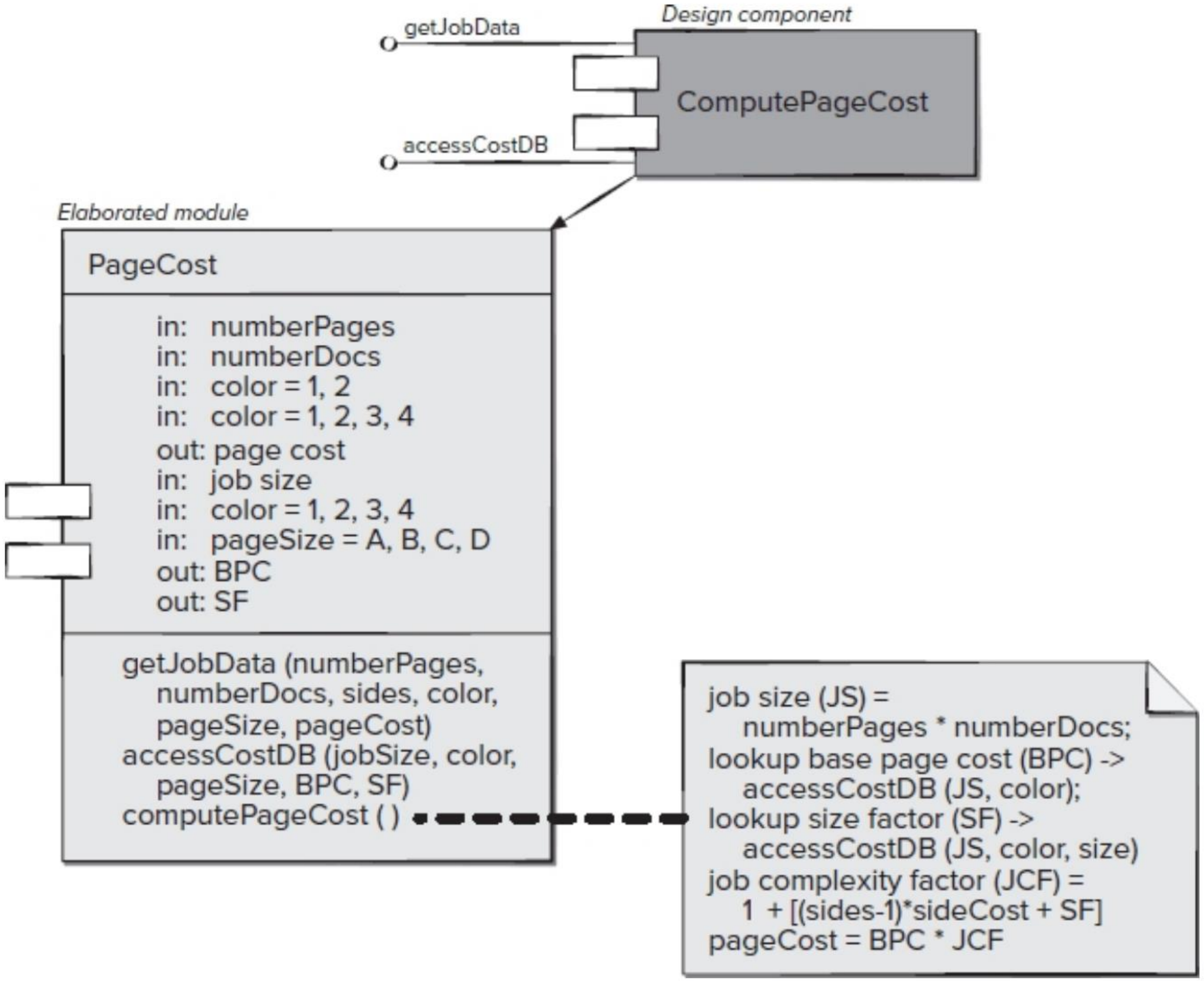

User
Web Application
Issuing Certificate Authority
Database
Third-party app
Certificate Issuer Gateway
Data Model
Web API
Certificates
Web server
Agent Scheduler
Third-Party Credentials
Interface Plugin
Certificate Locations
Remote Agent
Agents API
Platform

Data design

- Encompasses representation and manipulation of information in the problem domain.
- Common problems include:
  - Defining a database schema
  - Defining constraints or invariants regarding the data
  - Designing data interchange formats
  - Defining a module to represent data types and interactions with them
  - Designing data structures for algorithmic performance and/or visualization
  - Migrating/integrating data from disparate formats or data sets
- Facilitates component-level and interface design.

Component-level design

- Elaborates each component of high-level architecture and elements of the requirement class model.
  - Architecture ~ blueprint
  - Requirement model ~ "As a resident, I want to be able to shower"
  - Component design describes bathroom dimensions, layout, plumbing, and fixtures.
- Still assuming object-oriented program, representing components as *classes* with *properties* & *methods* in a *class diagram* of the component*.*
  - Typically more detailed than requirements classes, sometimes with pseudocode.
- Components include classes related to the problem domain (e.g. "certificate") as well as the infrastructure (e.g. "query", "button").
- Data types of input/output between elements must be specified as well.
- Structures for representing data collections should be defined.


Design component
getJobData
accessCostDB
ComputePageCost
Elaborated module
PageCost
in: numberPages
in: numberDocs
in: color = 1, 2
in: color = 1, 2, 3, 4
out: page cost
in: job size
in: color = 1, 2, 3, 4
in: pageSize = A, B, C, D
out: BPC
out: SF
getJobData (numberPages, numberDocs, sides, color, pageSize, pageCost)
accessCostDB (jobSize, color, pageSize, BPC, SF)
computePageCost ( )
job size (JS) = numberPages * numberDocs;
lookup base page cost (BPC) -> accessCostDB (JS, color);
lookup size factor (SF) -> accessCostDB (JS, color, size)
job complexity factor (JCF) = 1 + [(sides-1)*sideCost + SF]
pageCost = BPC * JCF

Interface design

- Describes how users interact with the system, as well as how the program interacts with other software, and how modules of the program interact with each other.
- Describes how the software "looks" to the end user.
- Significantly impacts user impressions of software quality and appeal.
- Depends on use cases and choices about architecture.
- Should not dictate architecture or infrastructure design; should be able to swap out one interface for another that does the same things.
- Occurs throughout design of other elements.
- Contemporary programs must support multiple interfaces (eg mobile)

User Experience (UX) Engineering

- The practice of making the program work in an intuitive, consistent, satisfying manner for users to accomplish their goals. Intersects cognitive science and human behavioral analysis.
- Typically includes screen layout, color scheme, fonts, aesthetics of interactive elements, animations, other media.
- Now includes voice commands, complex sensors and controllers, Augmented/Virtual Reality display, and other advanced interfaces.
- Must consider *accessibility* – the ability of users with impairments to use the program as easily as any other user. Common considerations:
  - Color blindness – Information should not be conveyed by color alone
  - Unsteady movement – Navigation should work with tab/arrows, not just mouse
  - Slow reading – Interactions should be self-paced (also helps non-native speakers or even just speakers on a slow connection).

Deployment design

- Defines where and how the software runs.
- May be (partially) specified in initial requirements.
- Results in a list of requirements and preconditions for deploying software.
- Fundamental constraints must be well-defined before implementation begins; constrains choices about architecture & component design (e.g. language).

Effective design principles

- Primitiveness – A method should do one task, and there should only be one method to do a task. Code should be reused rather than duplicated.
- Traceability – A method or other code snippet should be able to be traced back to the design of that component, and back to the requirement that led to that design.
- Symmetry – Design of components should be similar to each other within a single product.
- Dependency Inversion – Components should depend on abstractions, not implementations.
- Interface Segregation – Many narrowly-tailored interfaces between systems or modules are better than one all-purpose interface.
- Common Closure – Classes that change together, belong together.

UML

- Unified Modeling Language, or UML, was developed to support a prescriptive software development process ("Rational Unified Process"), now used in other contexts regardless of process model.
- A standard language for specifying, documenting, communicating and visualizing 13 types of software models.
- Usually more formal than required; informal diagrams can still easily capture design and provide flexibility with less work.
- May be automatically generated by an IDE or other visualization tool, or may be used as a basis for automating construction of a program skeleton, database, or documentation. Interoperability is the main benefit of following a rigorous standard.

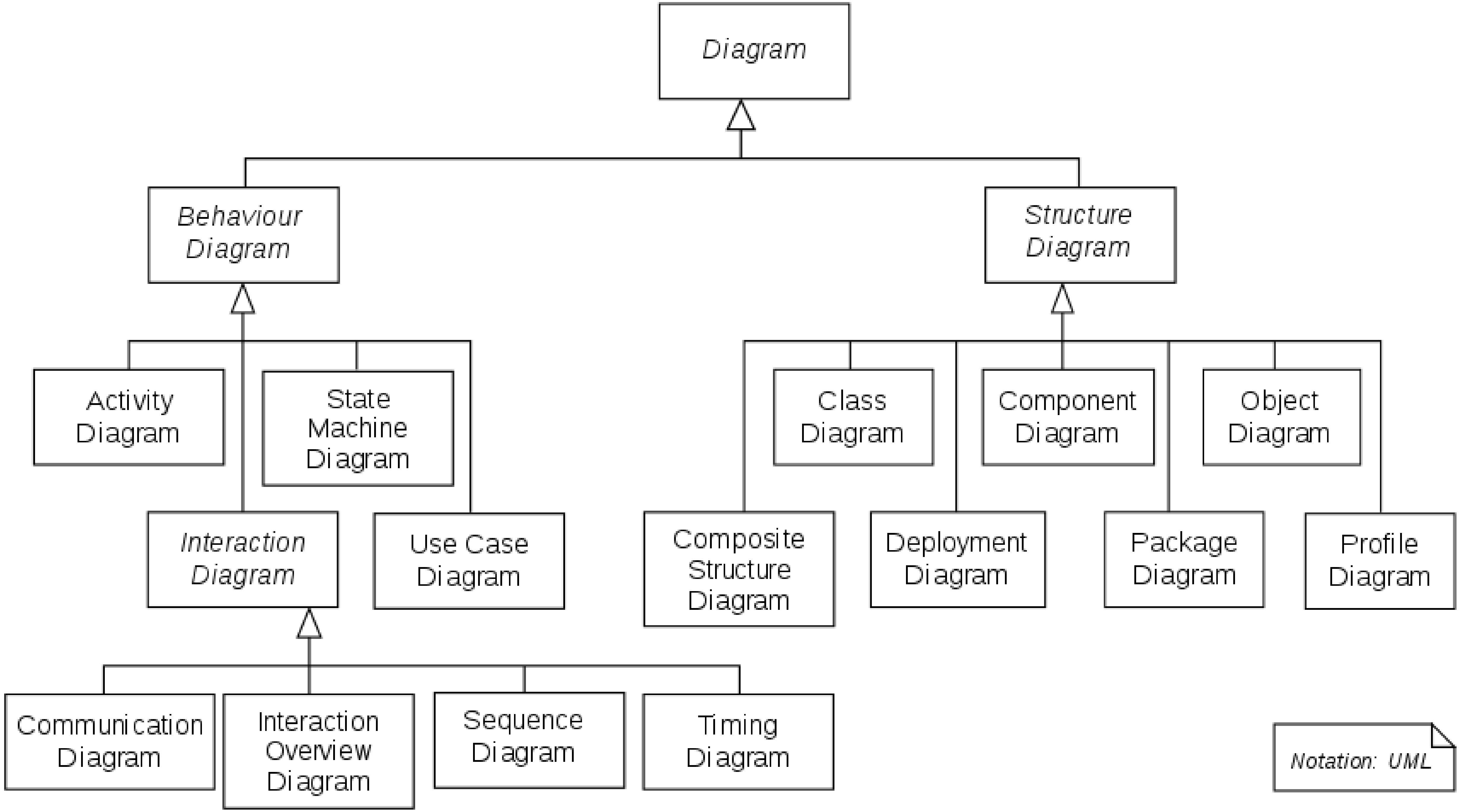


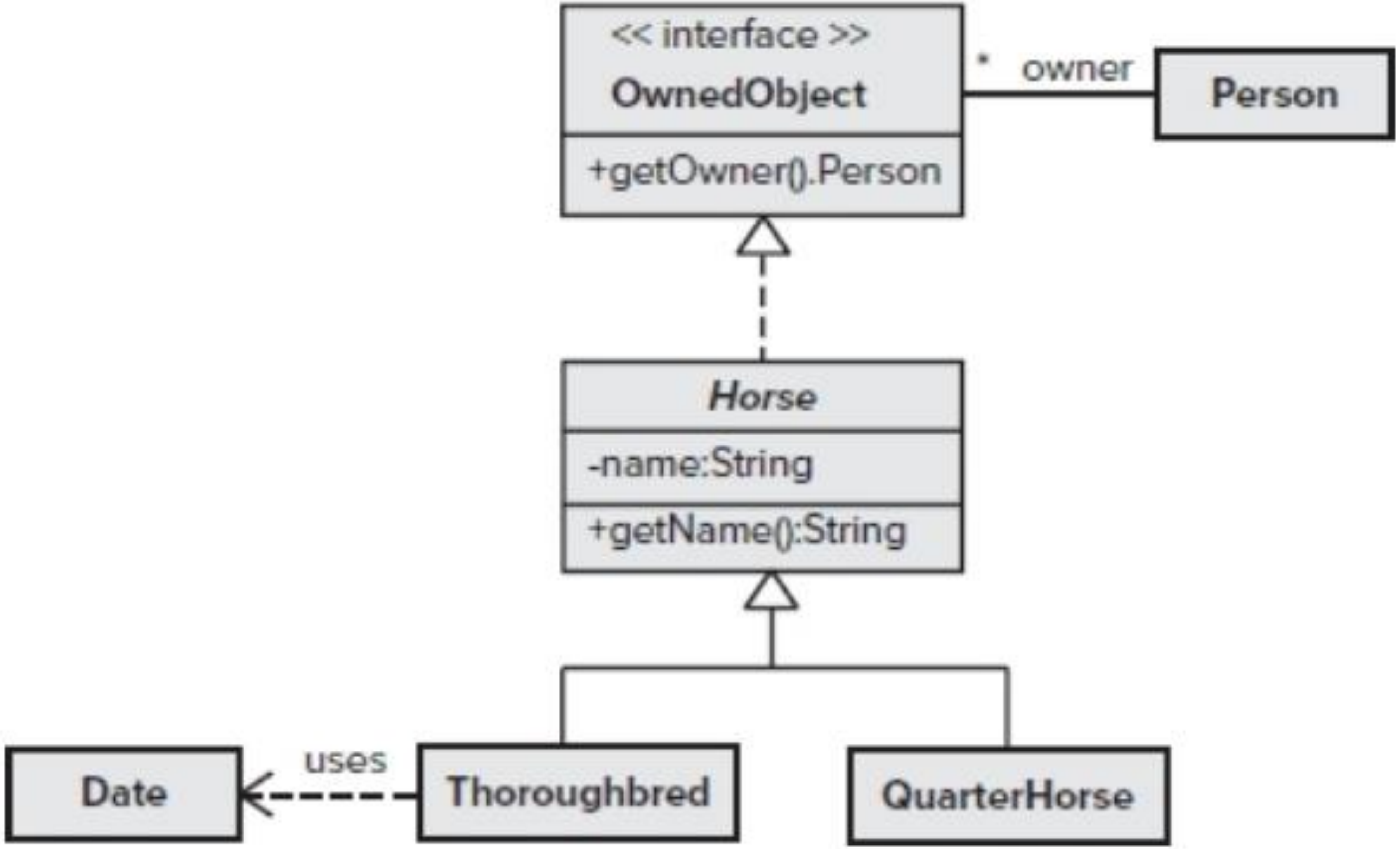
Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Interaction Diagram
Use Case Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Communication Diagram
Interaction Overview Diagram
Sequence Diagram
Timing Diagram
Notation: UML

Class diagram

- Models classes, attributes, operations/methods, and relationships.
- Foundation of Object-Oriented Programming.
- Captures a static view of the structure of a program.
- Does not capture communication or data flow.
- A box for each class or interface, divided into three parts:
  - Name on top. Abstract class names in italics. Interfaces start with "<<interface>>"
  - Attributes in middle, consisting of "<visibility><name>: <type>"
  - Operations on bottom; "<visibility><name>(<paramname>:<type>): <returntype>"
  - Visibility can be '-' (private), '#' (protected), '~' (package), or '+' (public)
  - Attributes and operations may be underlined to indicate *static* elements

Class diagram relationships

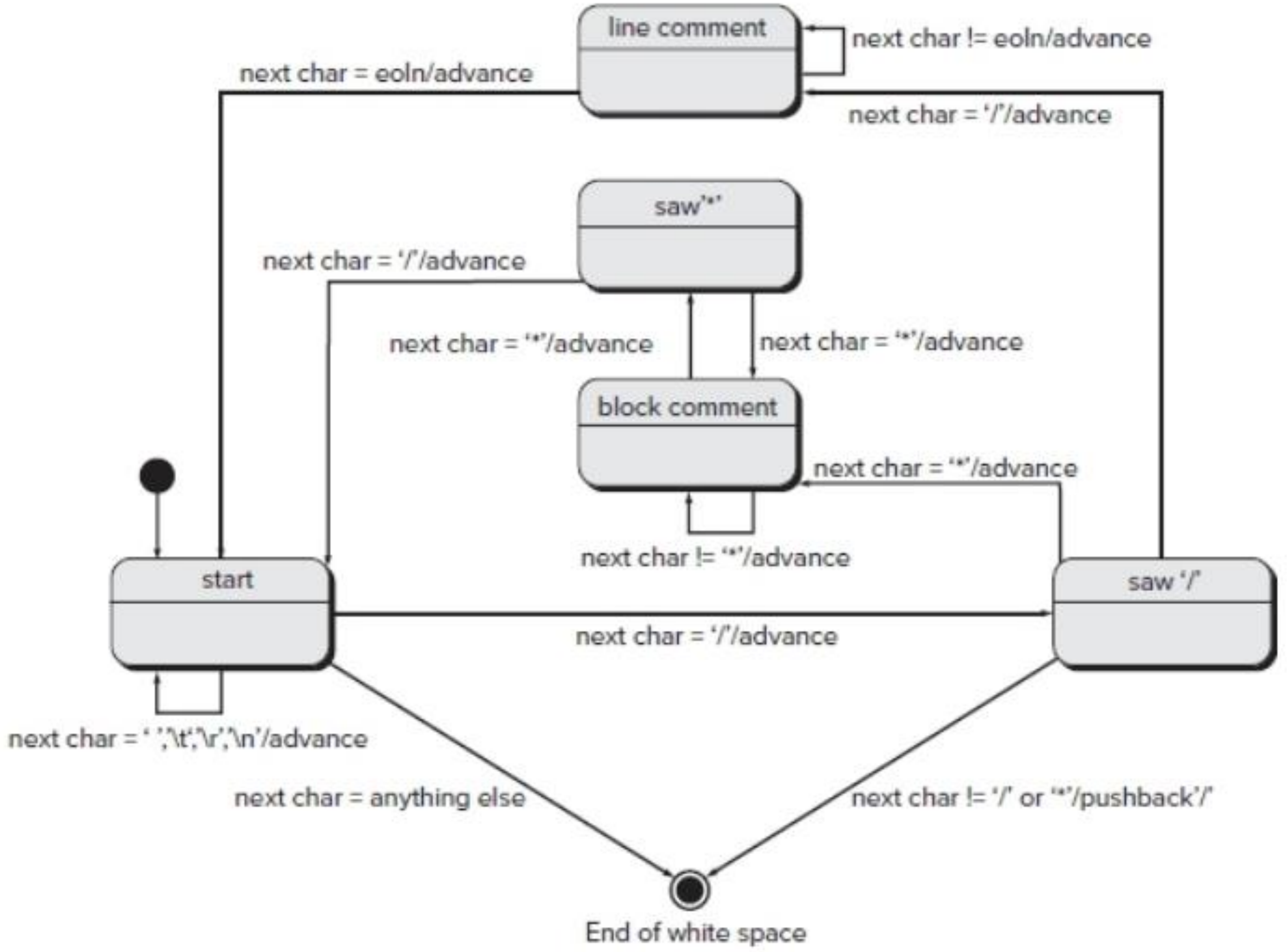
- Subclass has a solid arrow with a hollow triangle pointed at base class
- Implementation of an interface has dashed line with hollow triangle to the interface it implements.
- Solid line indicates association between classes. May be labeled. Angled arrows (<- or ->) indicate navigation. If a method of one class returns an object of another class, an association should be marked.
- Dashed lines indicate dependencies. A dependency is an association where a change to the depended class may require a change to dependent class.
- A number or number range on one end of an association indicates how many objects of that class may be associated with an object of the other.
- A diamond at one end of an association indicates ownership. Solid means the class on the other end is wholly owned by that class.



<< interface >>
OwnedObject
+getOwner().Person
*
owner
Person
Horse
-name:String
+getName():String
uses
Date
Thoroughbred
QuarterHorse

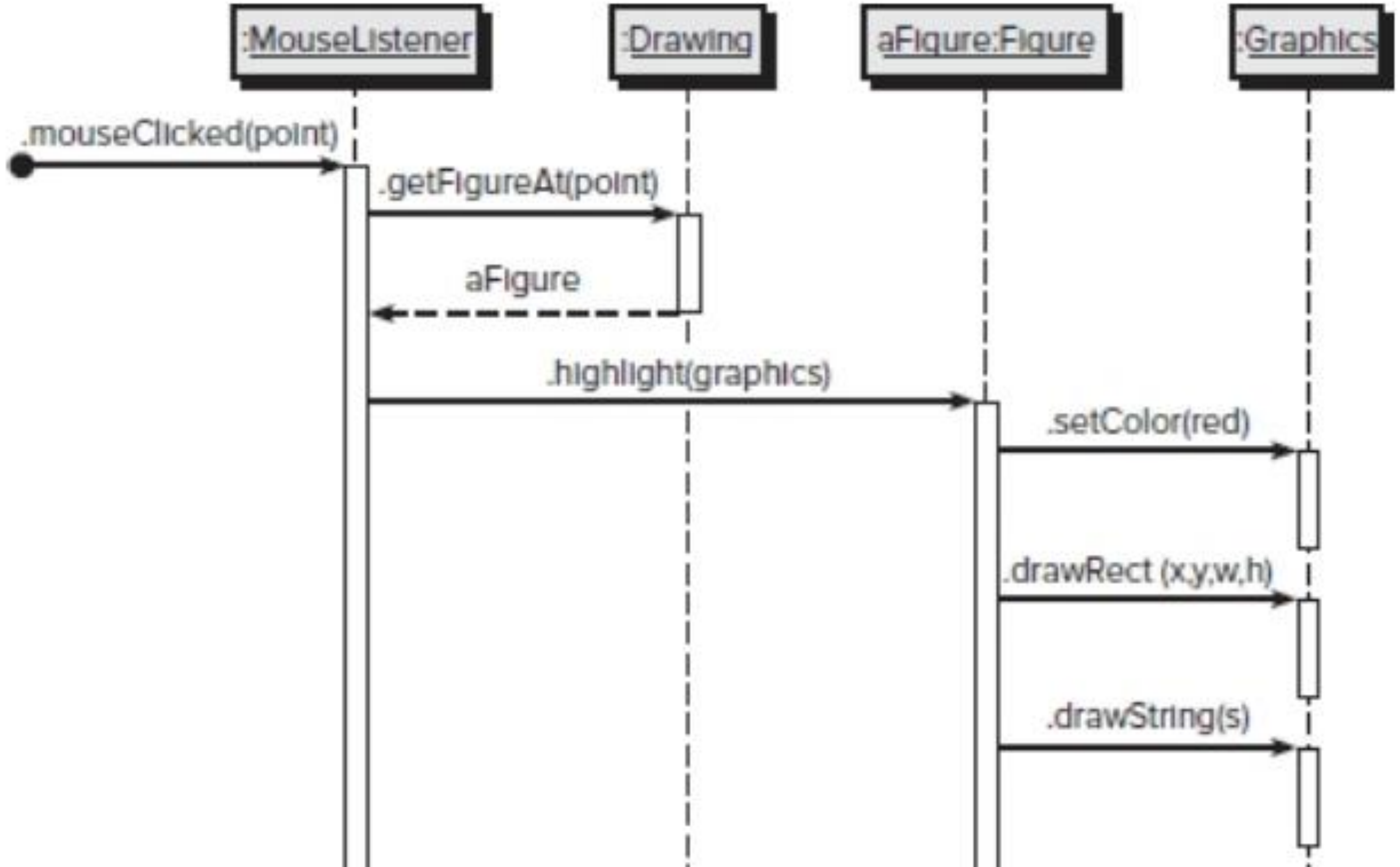
State diagram

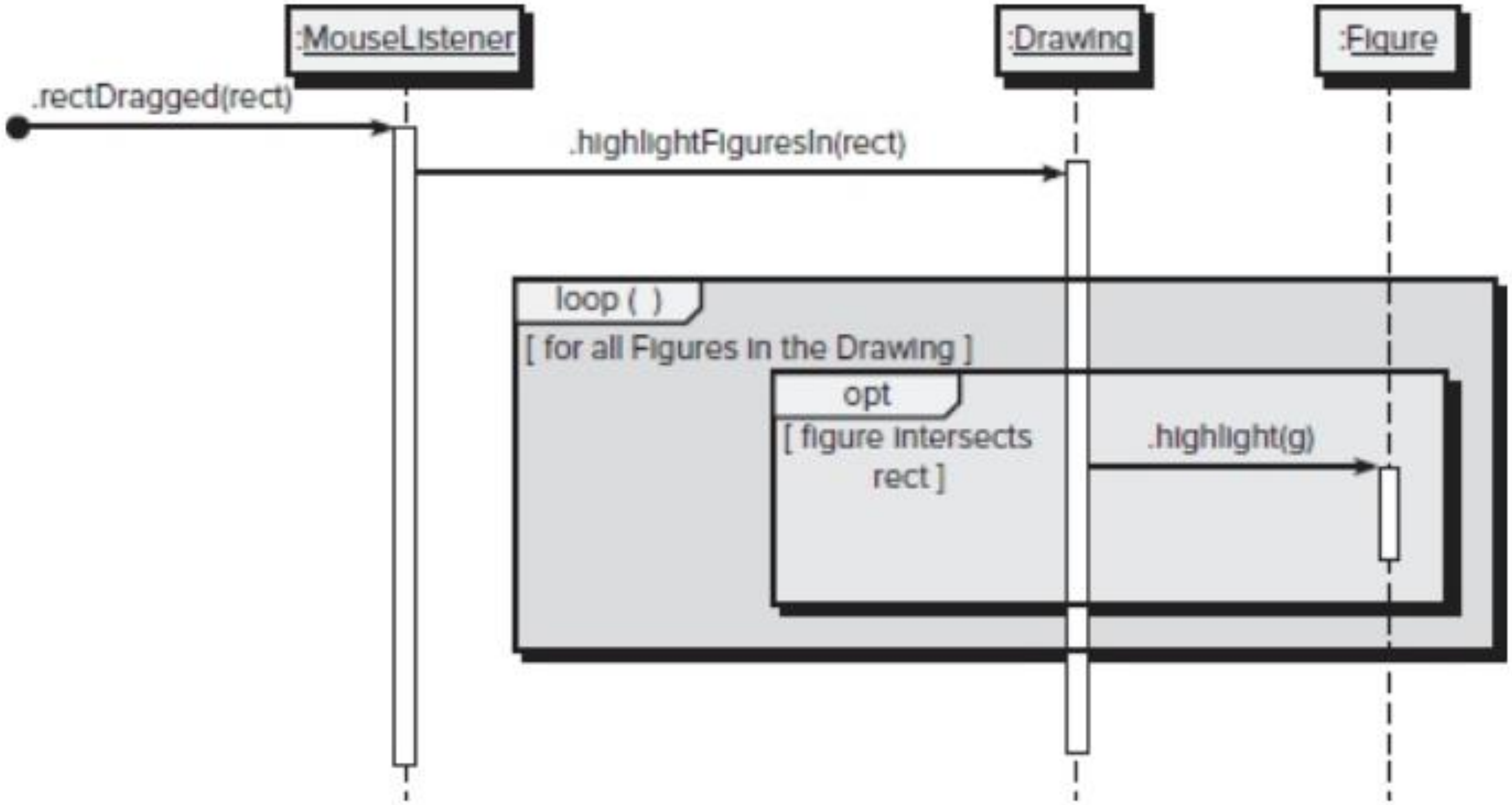
- Represents the possible states of a system or object with the actions that can be performed from each state and the state transitions.
- Rounded rectangle labeled with the name of each state in the top half, and any ongoing activity that occurs in that state on the bottom.
- Arrow pointing from one state to another indicates a transition between states. Labeled "condition / action (side effect)"
- One black circle points to the initial state. Another black circle with a white circle around it indicates a final state where no more transitions occur. Nothing points to the initial state or from the final state.


line comment
next char != eoln/advance
next char = eoln/advance
next char = '/'/advance
saw'*'
next char = '/'/advance
next char = '*'/advance
next char = '*'/advance
block comment
next char = '*'/advance
next char != '*'/advance
start
saw '/'
next char = '/'/advance
next char = ' ','\t','\r','\n'/advance
next char = anything else
next char != '/' or '*'/pushback'/'
End of white space

Sequence diagram (swimlane)

- Captures dynamic execution of program for one task or activity.
- Row of objects at the type show participants in the activity. A dashed "lifeline" from each object extends to the bottom of the diagram.
- Horizontal solid arrows from a *caller* lifeline to a *callee* lifeline indicate method calls, labeled with the method and parameters.
- Horizontal dashed line from callee to caller indicates a returned object from the caller's last unreturned method call to the callee.
- Time is read from top to bottom. An arrow should come below all calls that execute before it.
- Boxes around a section of the diagram indicate loops and conditionals
- Objects can execute methods on themselves with a "u-turn" arrow.
- Objects created during execution of the sequence start below the top row.


:MouseListener
:Drawing
aFigure:Figure
:Graphics
.mouseClicked(point)
.getFigureAt(point)
aFigure
.highlight(graphics)
.setColor(red)
.drawRect (x,y,w,h)
.drawString(s)


:MouseListener
:Drawing
:Figure
.rectDragged(rect)
.highlightFiguresIn(rect)
loop ( )
[for all Figures in the Drawing]
opt
[figure intersects rect]
.highlight(g)

References

- A Proposal for a Formal Definition of the Design Concept. Paul Ralph and Yair Wand. Jan 2009. Design Requirements Engineering: A Ten Year Perspective.
- User Story vs Use Case. Visual Paradigm. 2022.
- Essential Software Architecture, 2nd edition. Ian Gorton. July 2006. Springer.
- Design Patterns: Elements of Reusable Object-Oriented Software. Gamma, Helm, Johnson, and Vlissides. August 1994. Addison Wesley.
  - Authors known as the "Gang of Four". Classic text on software design.
- Software Architecture Diagram. James Freeman. July 2021. Edraw.
- Software Engineering-Data Design. 1000 Source Codes. 2015.