

Architecture Patterns in Practice

Gundimeda Venugopal

Topic Coverage

- ❖ What is a Software Architecture Pattern?
- ❖ Layered Architecture Pattern (N-tier Architecture)
- ❖ Client Server Pattern
- ❖ Event Driven Architecture Patterns
- ❖ Microservices Architecture Pattern
- ❖ Serverless Architecture Pattern
- ❖ Model View Controller Pattern
- ❖ Pipe-filter pattern
- ❖ Master-slave pattern
- ❖ Blackboard pattern

What is a Software Architecture Pattern

- ❖ The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [SEI]
- ❖ An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context.
- ❖ An architectural pattern is a proven structural organization schema for software systems
- ❖ Architectural patterns serve as a blueprint for the software systems.
- ❖ Architectural patterns are similar to Design patterns but have a broader scope.
- ❖ The architectural patterns address various issues such as:
 - Performance
 - High availability
 - Scalability
 - Security
 - Maintainability
 - Testing
 - Deployment
 - Technology Stack

Architecture Patterns vs Design Patterns

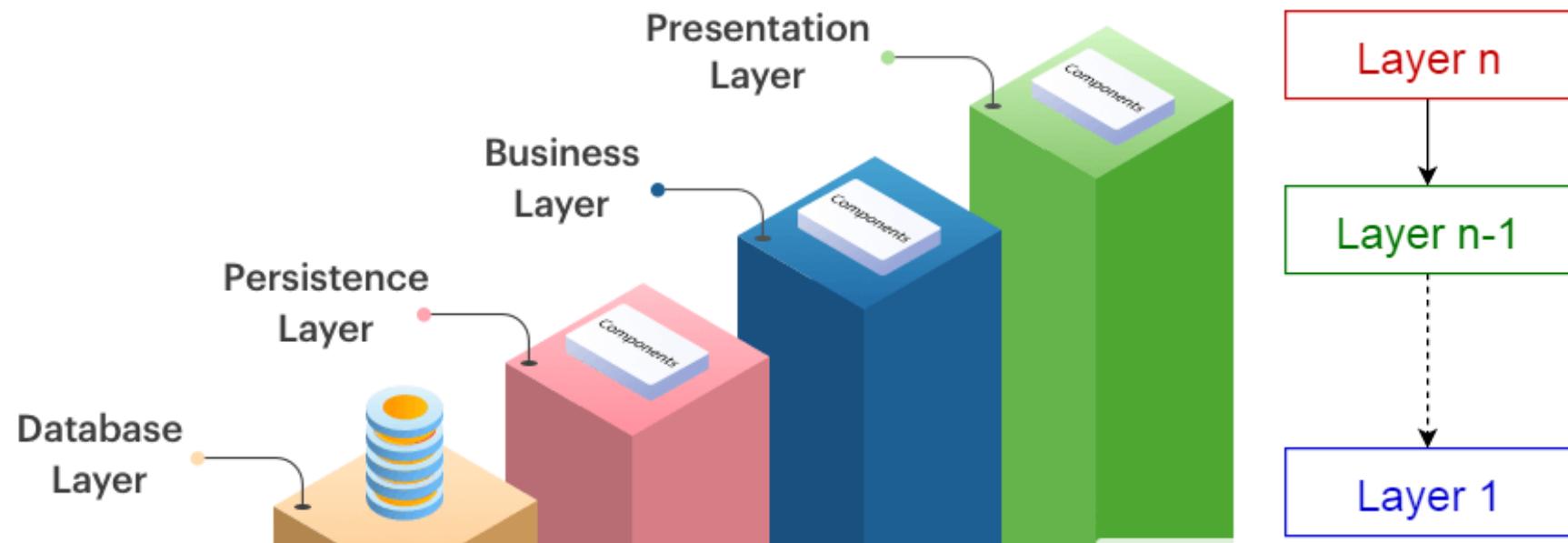
- ❖ The purpose of architecture patterns is to understand how the major parts of the system (e.g., components, modules or collection of classes) fit together, how messages and data flow through the system, and other structural concerns.
- ❖ Architecture represents scaffolding, the frameworks that everything else sits upon. Design patterns represent a way to structure classes to solve common problems. While both are designed to add clarity and understanding, they operate at different levels of abstraction.
- ❖ Architects must have both kinds of patterns at their disposal: Design patterns to build the best internal structure, and Architectural patterns to help define and maintain the underlying scaffolding of the application.

Architecture Patterns	Design Patterns
Definition	Fundamental structural organization for software systems
Role	Conversion of software characteristics to a high-level structure.
Example	Microservice, serverless and event-driven
Level	Large Level tool - concerns large scale components, global properties, and mechanism of the system
Problem Addressed	Distributed functionality, system partitioning, protocols, interfaces, scalability, reliability, security
	Specification that could help in implementation of a software
	Description of all the units of the software system to support coding
	Creational, structural and behavioral
	Small level tool- concerns schemes for refining and building smaller subsystems - structure and behavior of entities and their relationships
	Problems in software construction

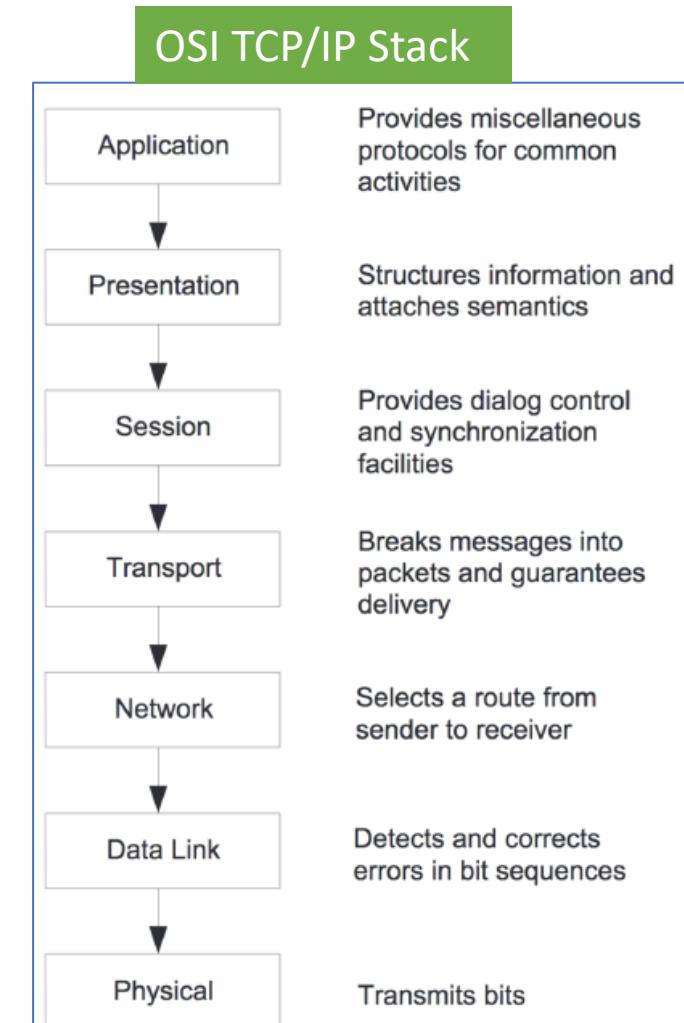
Architecture Patterns Usage Examples

#	Architecture Pattern	Practical Usage
	Layered Architecture (N-Tier)	4-Layer architecture (Presentation, Application/Service, Business Logic and Data access layers)
	Client Server	Desktop/Mobile Browser and HTTP Server running on port 80 Visual Basic/Mobile Application and a Server application Email client (e.g. Outlook) and an email server platform (e.g., Gmail, Hot mail)
	Master Slave	Database replication (Replication between a master database and one/more slave databases) Peripherals connected to a SPI / I2C bus in a computer system/microcontroller
	Pipe filter Pattern	Unix pipes Image processing pipeline Compilers: Consecutive filters perform lexical analysis, parsing, semantic analysis & code generation. NLP processing pipeline CI/CD pipeline
	Event Driven Architecture (Mediator, Broker)	Windows Event Model Java Observer and Observable ecosystem Object Request Broker (e.g., Orbix) Middleware (e.g., Kafka, WebSphereMQ, RabbitMQ, ActiveMQ ...)
	Blackboard	Global Logger for multiple processes (using shared memory on Unix) DeepQA Jeopardy Challenge

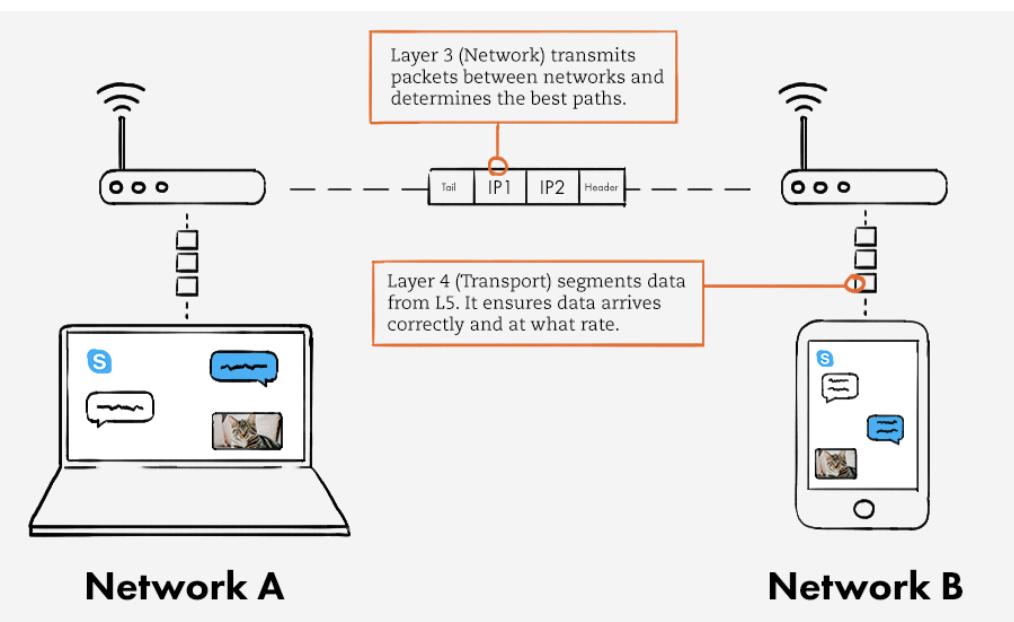
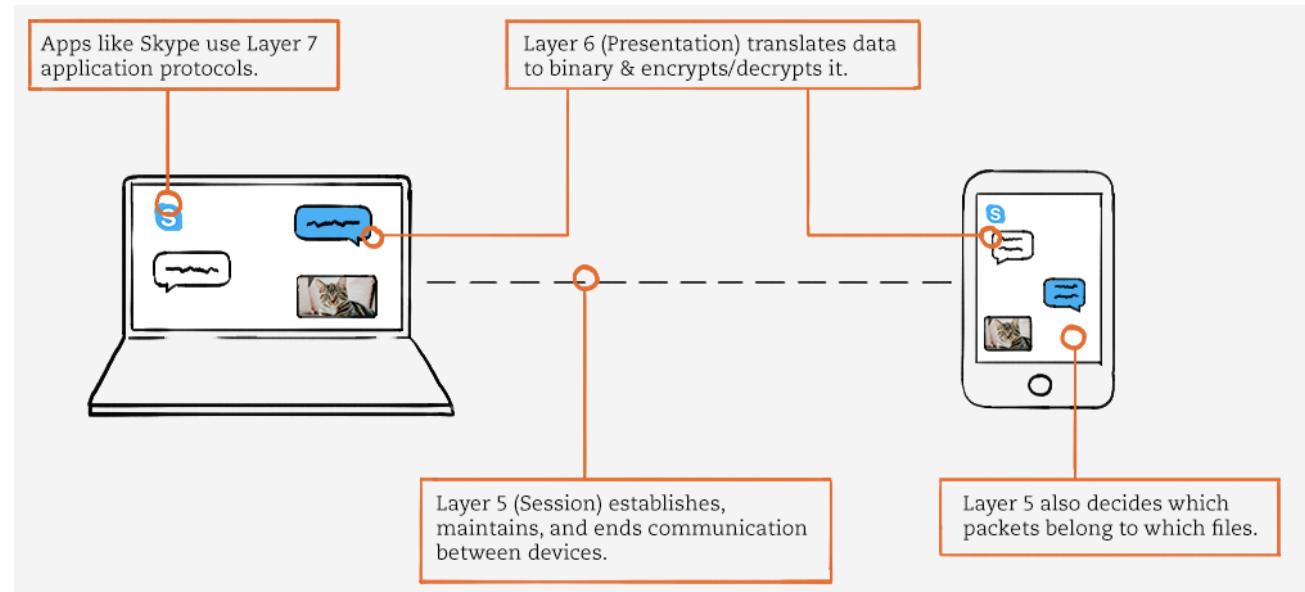
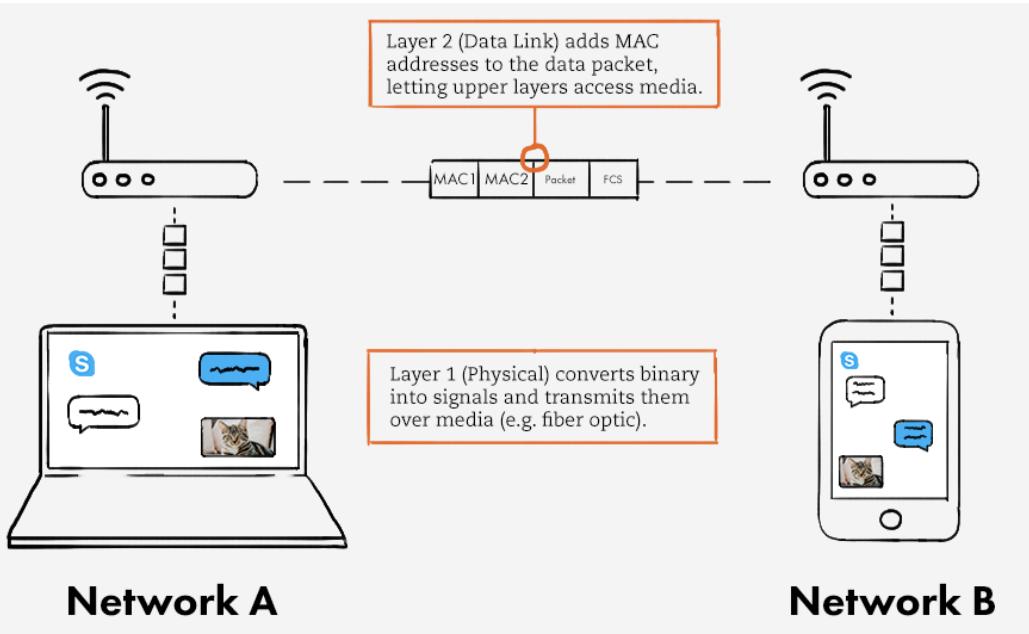
Layered Architecture pattern (N-tier)



- ❖ This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.
- ❖ Each layer plays a distinct role within the application and is marked as closed, which means a request must pass through the layer right below it to go to the next layer.
- ❖ layers of isolation – enables modification components within one layer without really affecting the other layers.
- ❖ A closed layered architecture means that any given layer (or tier) can only use the services of the next immediate layer, whereas in an open layer architecture a given layer can make use of any of the layers (or tiers) below it.
- ❖ Closed layer architecture promotes independence between layers by reducing dependencies.



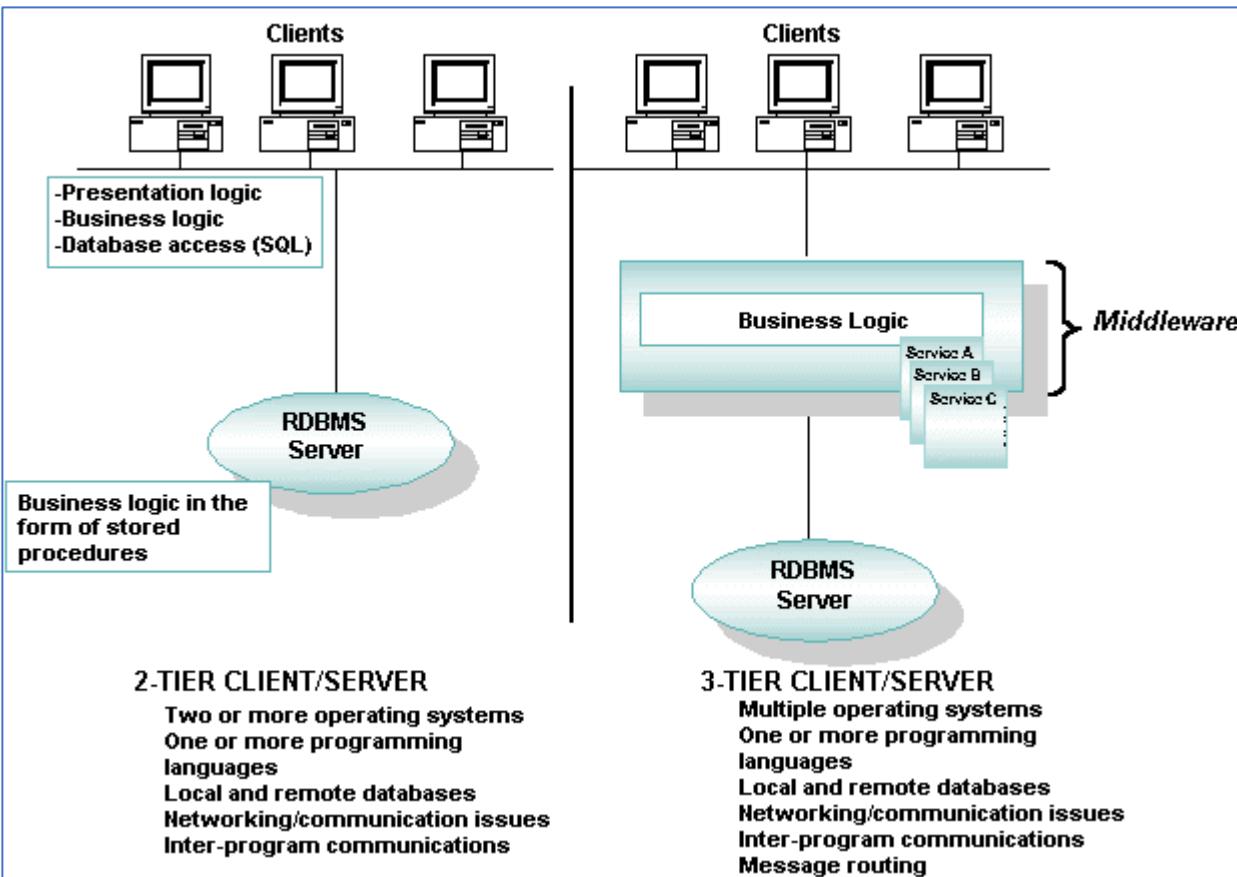
Layered Architecture Example – TCP/IP



- ❖ Layer 1 – Physical: Converts binary into signals and transmits them over media (e.g. fibre optic)
- ❖ Layer 2 – Data link (MAC Layer): Add MAC address to the data packet letting upper layers access media. Detects and corrects errors in bit sequences
- ❖ Layer 3 – Network (IP Layer): Transmits packets between networks. Select the route from sender to receiver and determines best paths
- ❖ Layer 4 – Transport(TCP layer): Segments messages into packets and guarantees message delivery
- ❖ Layer 5 – Session: Establishes, maintains and ends communication between end devices
- ❖ Layer 6 – Presentation: Translates data to binary and encrypts/decrypts it
- ❖ Layer 7 – Application: Apps like skype, browser, ftp client use Layer 7 application protocols

Client Server Architecture Pattern

- ❖ In client/server architecture, clients, or programs that represent users who need services, and servers, or programs that provide services, are separate logical objects that communicate over a network to perform tasks together.
- ❖ A client makes a request for a service and receives a reply to that request; a server receives and processes a request, and sends back the required response.



Every client/server application contains three functional units:

- ❖ Presentation logic or user interface (for example, ATM machines)
- ❖ Business logic (e.g., interest calculation and Balance retrieval logic)
- ❖ Data (for example, records of customer accounts)

These functional units can reside on either the client or on one or more servers in your application. Which of the many possible variations you choose depends on how you split the application and which middleware you use to communicate between the tiers.

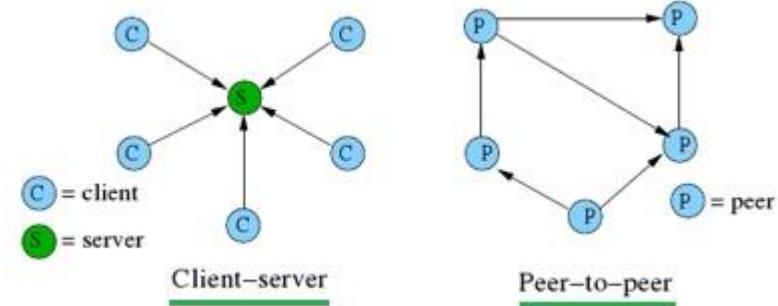
In 2-tier client/server applications, the business logic is buried inside the user interface on the client or within the database on the server in the form of stored procedures. Alternatively, the business logic can be divided between the client and server. File servers and database servers with stored procedures are examples of 2-tier architecture.

In 3-tier client/server applications, the business logic resides in the middle tier, separate from the data and user interface. In this way, processes can be managed and deployed separately from the user interface and the database. Also, 3-tier systems can integrate data from multiple sources.

- ❖ In Client Server architecture, The server contain only a limited no of resources(Ram, CPU, storage,..etc.). When no of users increases its hard to handle all client requests with single server. sometimes the server processor utilized and unable to give the service

Client Server Architecture Pattern

- ❖ **Client server architecture** model has two different modes or types. There are different types of architectures based on client server model viz. distributed, peer to peer etc

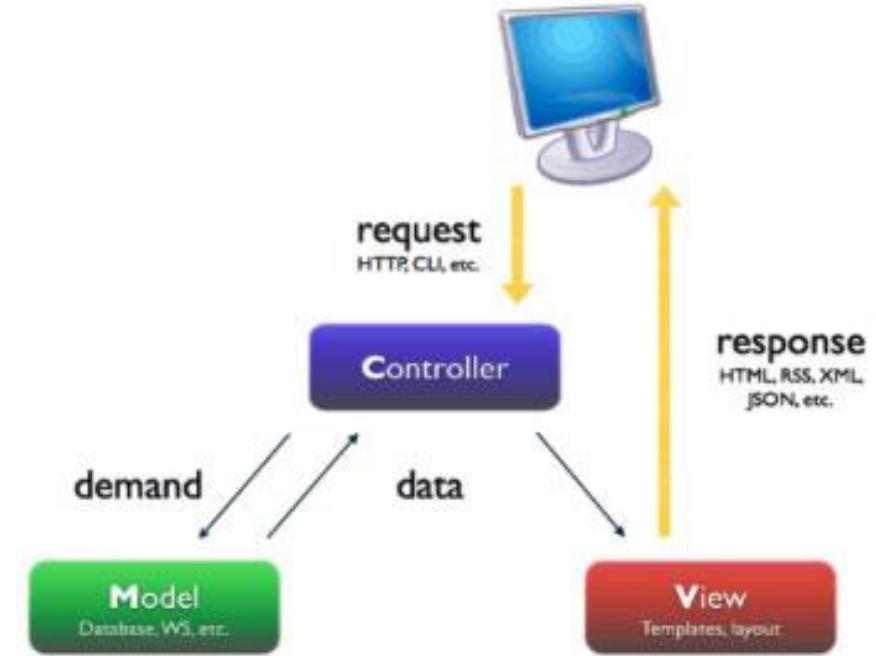
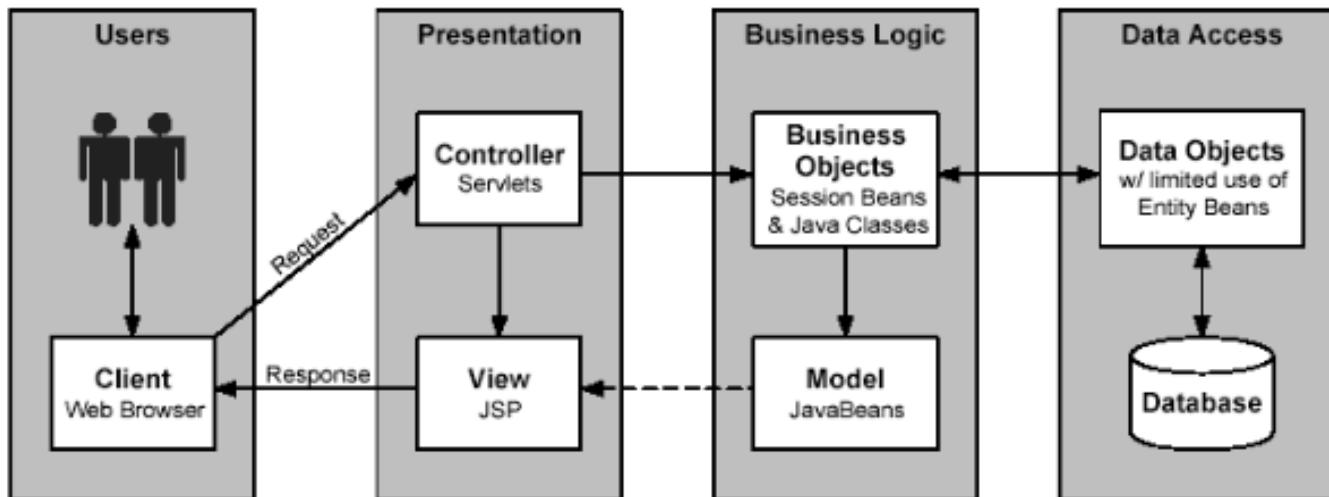


- ❖ In distributed architecture, one or more dedicated machines are used only as server while all the other machines are used as clients. In this scenario, clients can communicate via server. In this mode, client initiates communications. Client issues request to a server. Server replies or performs some service. Examples: 1) A Visual Basic app that talk to a database. 2) FTP client uploading/downloading to/from a FTP server. 3) A browser downloading web pages and uploading data to a HTTP server. 4) Signiant – faster file transfer. Fox used it to distribute 6TB of Avatar movie to Russia to prevent piracy. 5) CDNs such as Akamai, Limelight Networks, Amazon Cloud Front.
- ❖ In peer to peer (P2P) architecture, Each of the host or instance of application program can function as both client and server simultaneously. Both of them has equivalent responsibilities and status. In this mode, any participant can initiate communication. Any device can generate a request. Any device may provide a response. P2P examples: Skype, Zoom, BitTorrent (faster downloads), KaZaA (file sharing) and p2p CDN (STRIVECAST, peer5, now even akamai)

Model View Controller Architecture Pattern

The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, the view, and the controller.

- ❖ **Model** - Represents the data of the application. This matches up with the type of data a web application is dealing with, such as a user, video, picture or comment. Changes made to the model notify any subscribed parties within the application.
- ❖ **View** - The user interface of the application. Most frameworks treat views as a thin adapter that sits just on top of the DOM. The view observes a model and updates itself should it change in any way.
- ❖ **Controller** - Used to handle any form of input such as clicks or browser events. It's the controller's job to update the model when necessary (i.e. if a user changes their name in a web page).



Client	Web app
View	JSP, Velocity, FreeMarker, JSF, Tiles,
Controller	Servlets
Model	Java Beans
Data sources/ Data Access	Different data sources SQL DB, No SQL DB, XML, JSON, List, file, table, ...
DB Frameworks	JDBC, Hibernate,, Ibatis

Model View Controller Benefits

❖ 1. Faster App development due to separation of concerns

MVC supports parallel development of apps. In MVC, a developer can write code for the View while another developer can work on Controller to write business logic for the application. A team can perform multiple tasks at a time that leads to rapid app development.

❖ 2. Provide multiple views:

MVC allows developers to build multiple views for a model easily.

❖ 3. It Supports asynchronous technique:

The model of MVC is compatible with JavaScript Frameworks. Therefore, MVC based apps work well with site-specific browsers, PDF files, as well as with desktop widgets. Support to asynchronous technique further allows developers to build apps that load faster.

❖ 4. Modification does not affect the entire model:

As the Model part does not depend on the views to process data, changes in either section do not affect the entire model. Due to this, MVC comes handy to develop apps that may need frequent changes in the user interface like fonts, colours, screen layouts and support for mobile or tablets, etc.

❖ 5. The Architecture works with any type of data:

Model-View-Controller architecture returns data without formatting. It allows using the same components in any interface. Take for example, developers format data into HTML, and with MVC, they can access data in formats for Macromedia Flash or Dream-Viewer.

Microservices Architecture

- ❖ Microservices Architecture is “Service – oriented architecture composed of loosely coupled elements that have bounded contexts”
 - Service oriented architecture is all about services that communicate with each other over the network
 - Loosely coupled services allow update of services independently; updating one service does not require changing any other services
 - Bounded contexts mean microservices are self contained. You can update the code without knowing anything about internals of other microservices
- ❖ Microservices are about problems at scale: # of developers, # of features and # of users
- ❖ Microservices: How to decompose (how to identify service boundaries)
 - Define services corresponding to business capabilities. A business capability is something a business does in order to provide value to its end users.
 - High business cohesion: Things that change together, stay together
 - Loose Coupling (Implementation / Domain/ Temporal coupling)
- ❖ Microservices Embrace



Amazon

Thousands of teams

× Microservice architecture

× Continuous delivery

× Multiple environments

= 50 million deployments a year

(5708 per hour, or every 0.63 second)

Principles of Microservices

1. Rely only on the public API

- Hide your data
- Document your APIs
- Define a versioning strategy

2. Use the right tool for the job

- Polygot persistence (data layer)
- Polyglot frameworks (app layer)

3. Secure your services

- Defense-in-depth
- Authentication/authorization

4. Be a good citizen within the ecosystem

- Have SLAs
- Distributed monitoring, logging, tracing

5. More than just technology transformation

- Embrace organizational change
- Favor small focused dev teams

6. Automate everything

- Adopt DevOps

Microservices – Example Benefits and Disadvantages

❖ Example: Online Shopping application

- User Account Management
- Inventory Management
- Shipping Management
- Product Catalogue Management
- Order Management
- Product Recommendations
- Product Reviews Management

Each service can be owned by a different team who becomes an expert in that particular domain and an expert in the technologies that are best suited for those particular services. This often leads to more stable API boundaries and more stable teams.

❖ Benefits

- Microservices follow Single Responsibility Principle: “Do one thing and do it well”
- Organisational alignment
- Release functionality faster and Independent scaling
- Technology diversity; Adopt technologies faster
- Enable resiliency by designing for failure
- Enable security concern segregation

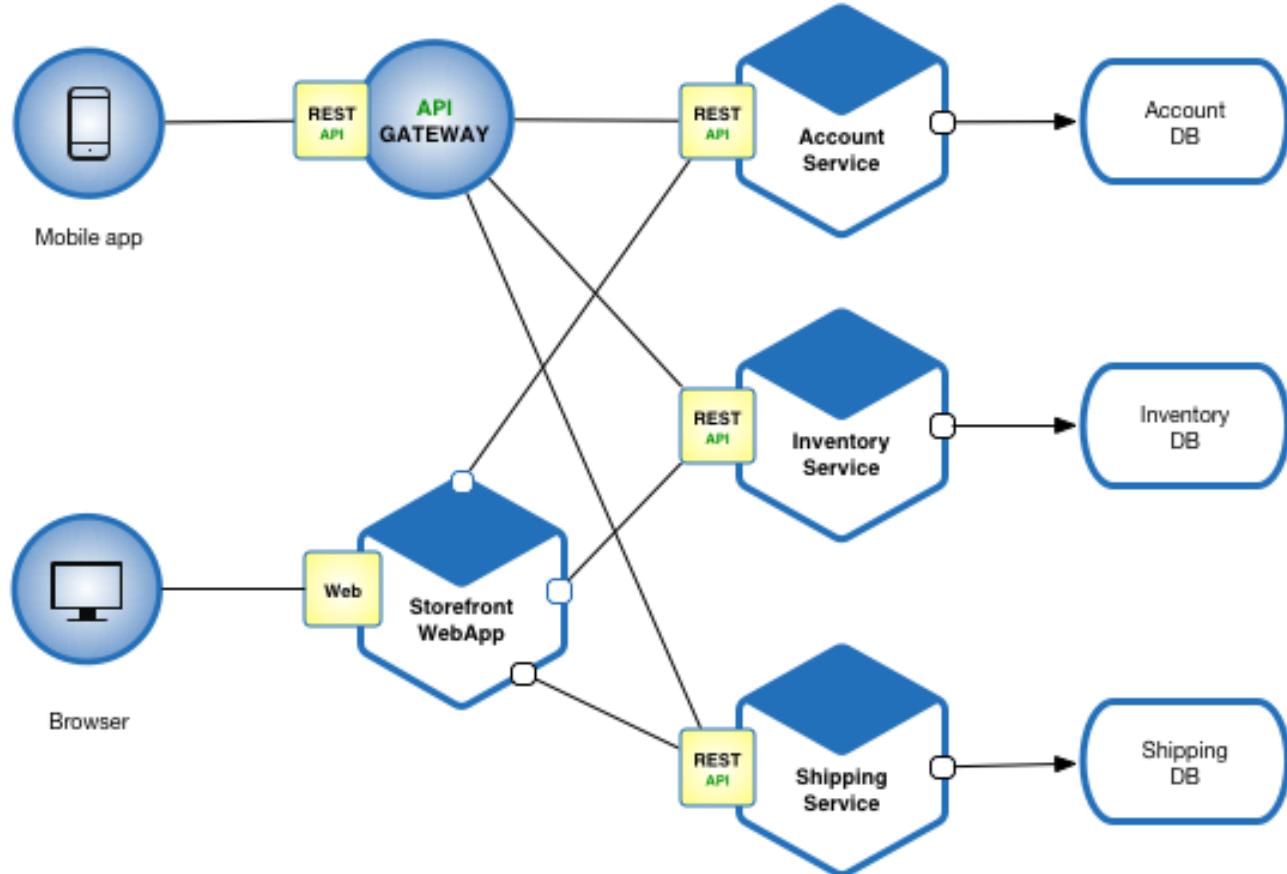
❖ Disadvantages

- Cognitive overloading (many tooling options / System understanding)
- Testing is more complicated
- Monitoring is more complex
- Operational overhead
- Resiliency is not free

Microservice Architecture: Online e-commerce application

Building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them.

The application consists of several components including the Store Front UI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders. The application consists of a set of services



Example: Online eCommerce application
User Account Management
Inventory Management
Shipping Management

Serverless Architecture Pattern

Serverless architectures refer to applications that significantly depend on third-party services (known as Backend as a Service or "BaaS") or on custom code that's run in ephemeral containers (Function as a Service or "FaaS")

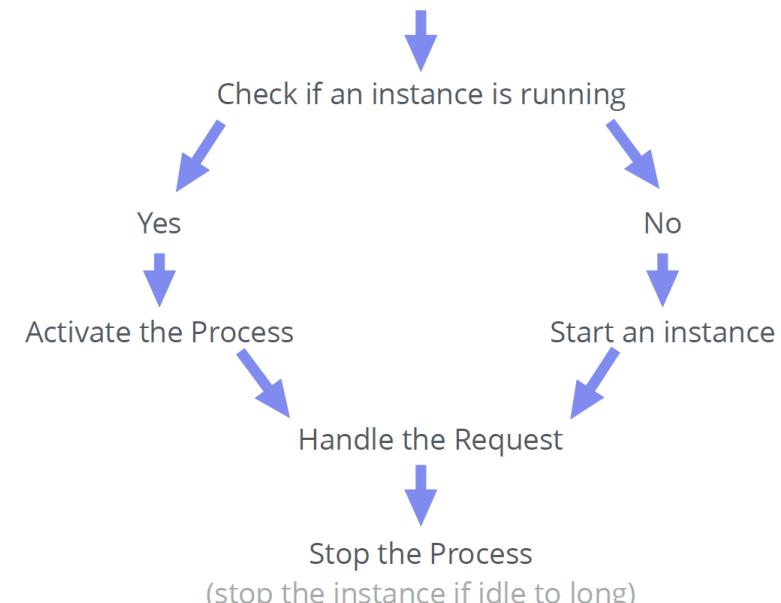
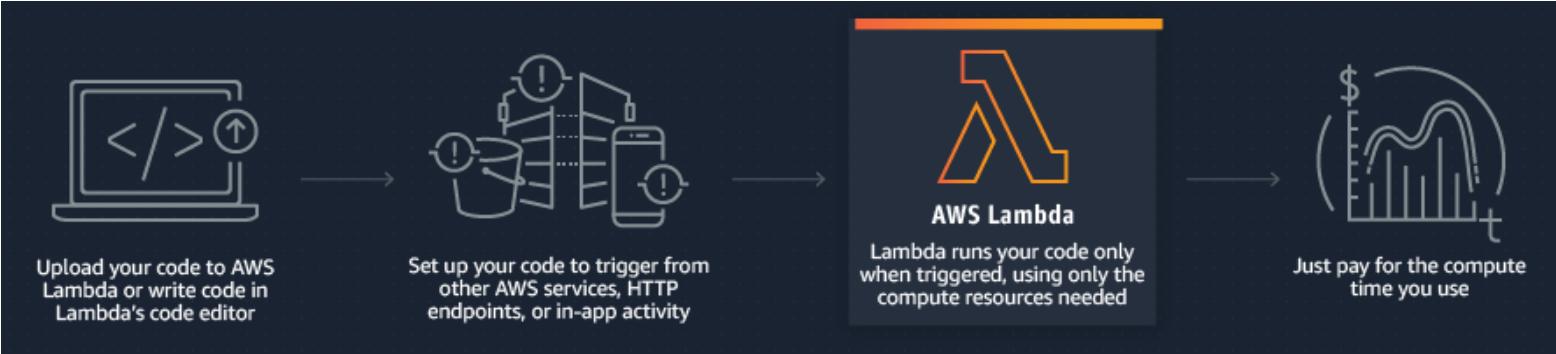
Write your code and run your applications without provisioning or managing anything on the server side.

- ❖ Serverless in a cloud environment means
 - Function as a unit of application logic
 - No Servers to provision or manage
 - Scales with usage
 - Pay for what you use and Never pay for idle
 - Built in availability and fault tolerance
 - Decreased time to market

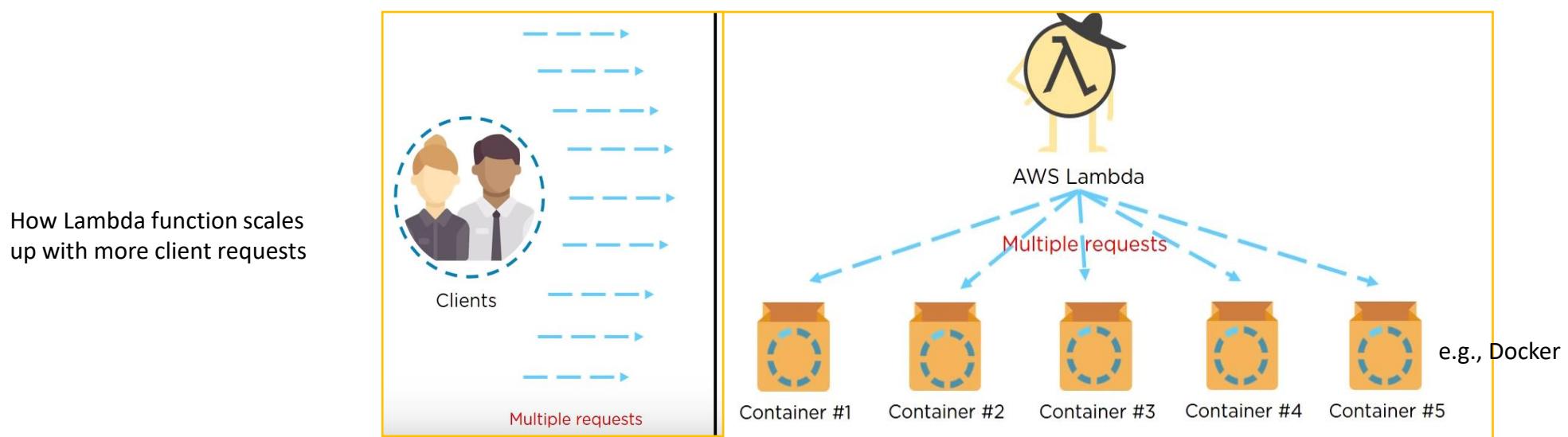
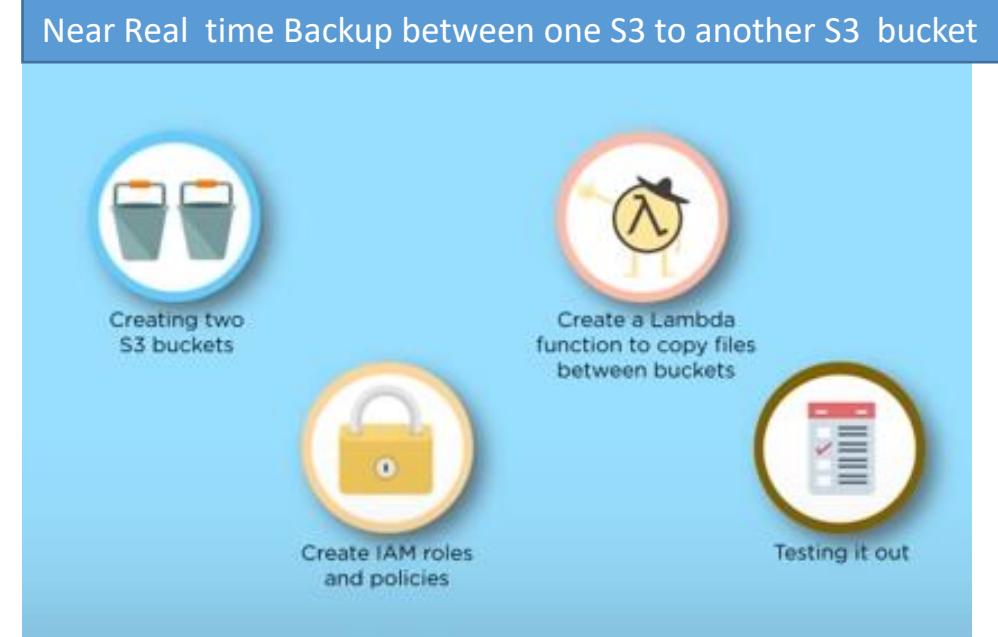
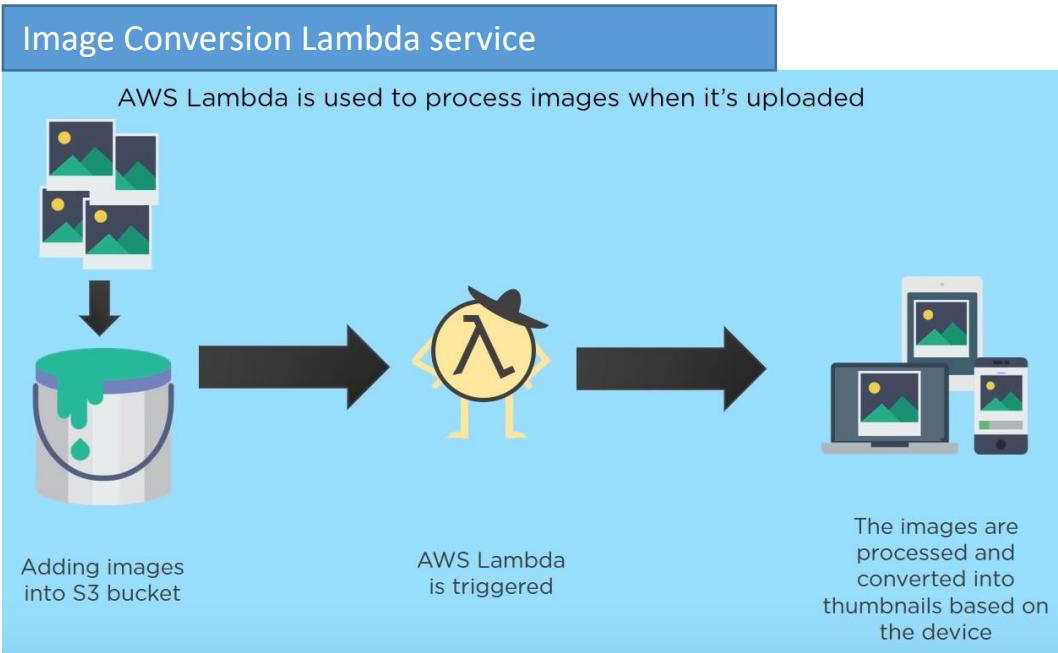
- ❖ How does it work?
 - Function as a unit of application logic
 - New invocation based on demand
 - Events trigger a function

- ❖ Serverless Providers
 - AWS Lambda
 - Google Cloud Functions
 - Microsoft Functions
 - IBM Openwhisk

- ❖ Use cases: Serverless Website | App Authentication | Mass Emailing | Real-time Data Transformation | CRON Jobs | Chatbot | IoT | image conversion | file compression
- ❖ Industry: Reuters Processes nearly 4000 requests/sec, Coco Cola,
- ❖ Supports Python, Go, Java, C#, NodeJS, ...



Serverless lambda function examples



Event Driven Architecture Pattern

- ❖ **Event-driven architecture (EDA)** is a software architecture paradigm promoting the production, detection, consumption of, and reaction to events.
- ❖ An *event* can be defined as "a significant change in state"

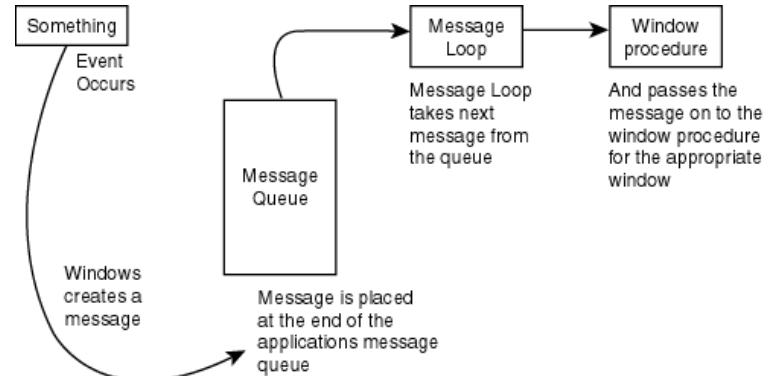
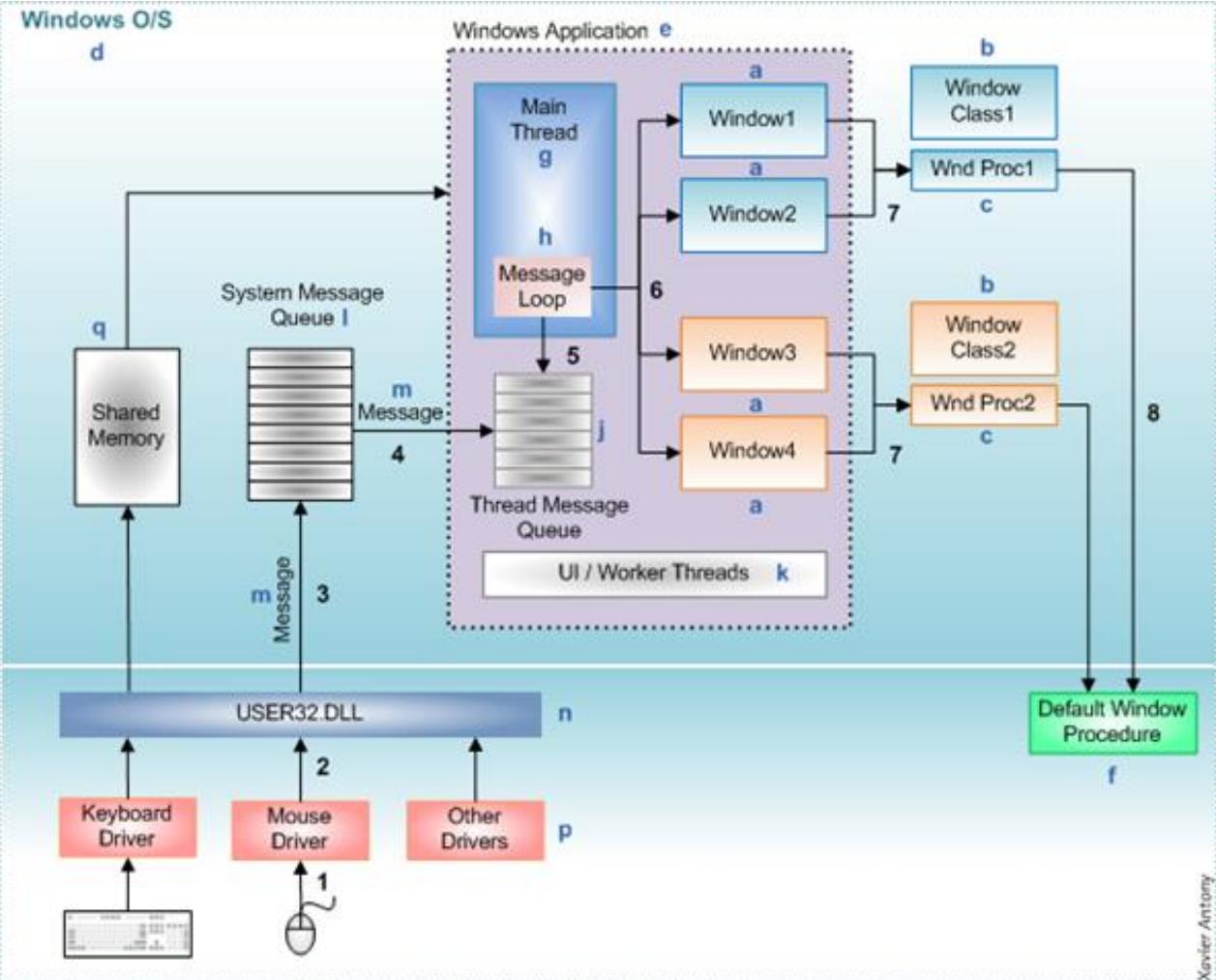
Examples:

- ❖ You're in the lobby of a building but you need to be on the 3rd floor. You press the button that calls the elevator which then travels down to meet you, opens its doors, welcomes you in, and then takes you to the floor of your choice where it opens up and lets you out. Pressing the button produced the event. The event listener, the elevator's operational system, processed the request sending it to the event consumer, the elevator car itself which then processed the trip from the ground floor to the 3rd. Nothing at all happened until the event of pushing the button happened.
- ❖ "Things" in the IoT tend to fall into two categories, sensors that detect things happening, and switches that enable responses to make other things happen. For example, a thermometer may detect that the temperature has risen above 80° F. in a given space. Exceeding that threshold is the event. Turning on the air conditioning is the result.
- ❖ Framework examples: Java AWT, Java Swing, Windows MFC

Advantages:

- ❖ Event Driven Architecture (EDA) processes are highly decoupled and can be scaled independently, which makes it a good option for modern, distributed and cloud-enabled applications that are horizontally scalable and resilient to failure.
- ❖ EDA aligns well with the increasing popular use of microservice architectures. Events never travel, they just occur. Everything else needs to be portable. The more each service can stand alone, the more resilient and fault-tolerant the system is
- ❖ EDA supports both asynchronous and distributed architecture.
- ❖ Event-Driven Architecture (EDA) creates an environment in which there's no waiting for requests. None of the event producers or consumers are aware of each other and remain so even after the transaction is completed.
- ❖ Given that EDA is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events, it is also highly adaptable and can be used for both small and large, complex applications.

Windows Message Loop

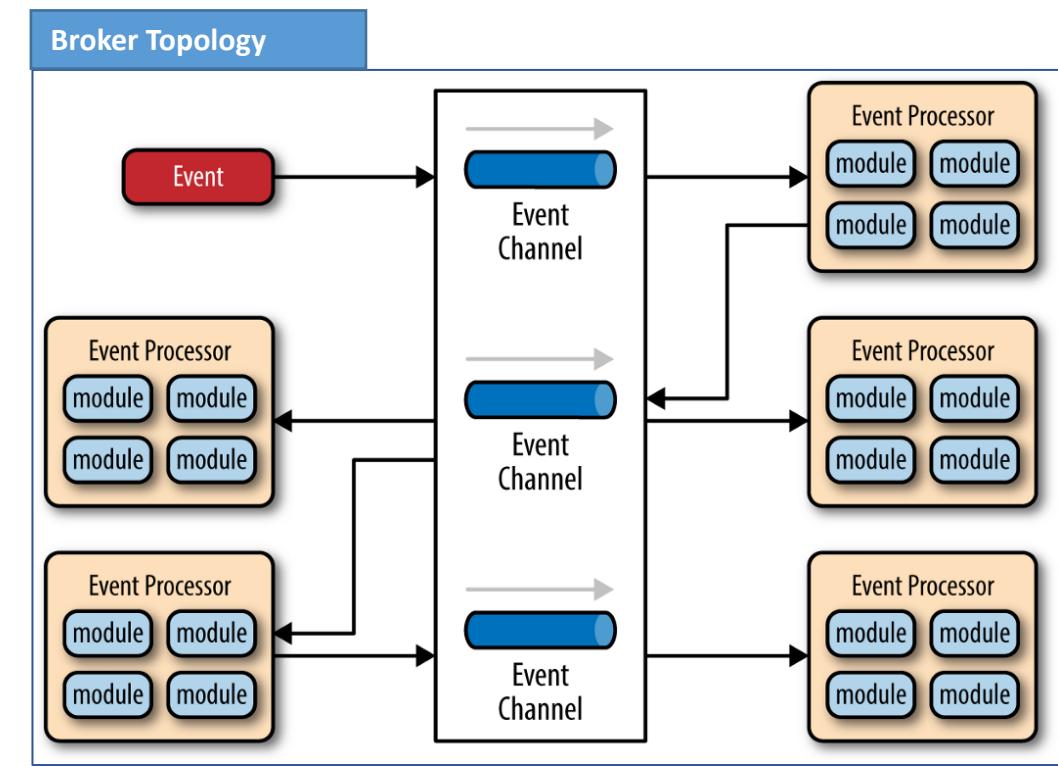
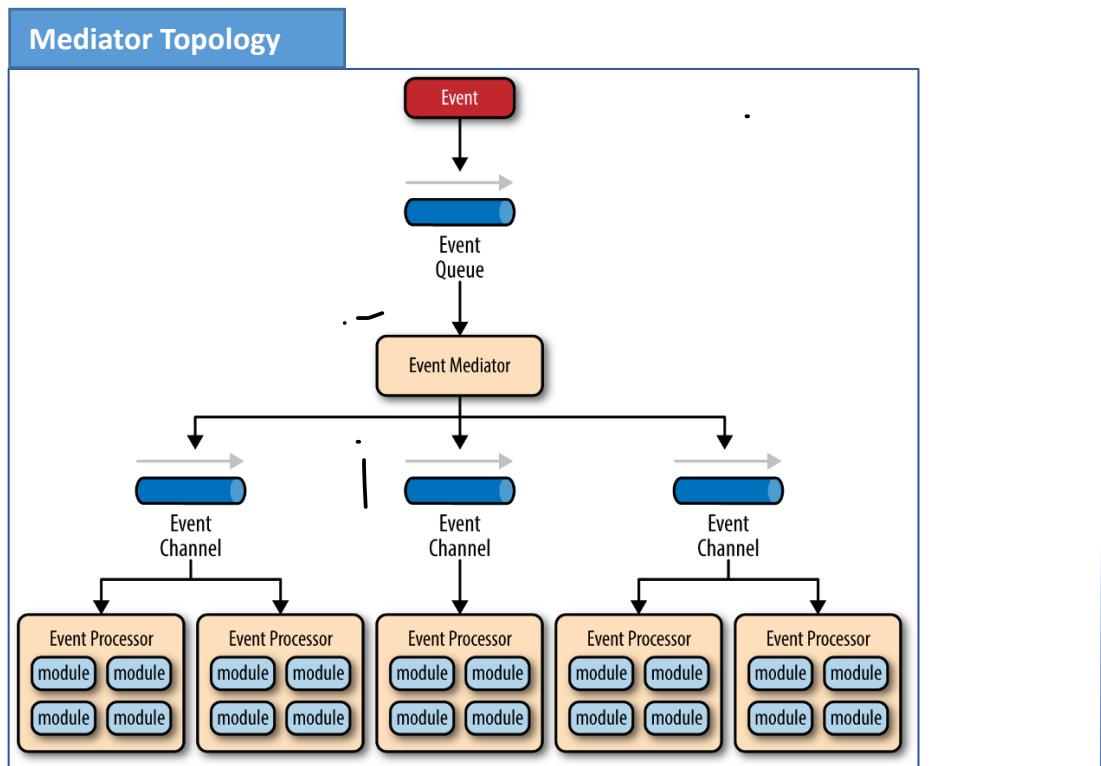


- ❖ When an application is started, the operating system creates a process and begins executing the primary thread (which has the start-up code entry-point **WinMain()** function) of that process. When this thread terminates, so does the process.
- ❖ The application then proceeds to create windows for the Graphical user interface. In the window class registration process, windows are associated with a **Window procedure** *through which you can specify what the window displays and how the window is to respond to user input by determining how the window will respond to system messages.*
- ❖ Keyboard and mouse activity cause the operating system to generate messages and send them to the proper application.
- ❖ The primary task of a Windows-based application can be seen as the processing of the messages that it receives. This processing involves the routing of messages to their intended target windows and the execution of expected responses to those messages according to the messages' types and parameters.
- ❖ It is the task of the application developer to map these messages to the functions that will handle them and to provide an appropriate response to the message..

Under MS-DOS, users enter command-line parameters in order to control how an application runs. Under Windows, users start the application first, and then Windows waits until users express their choices by selecting items within a graphical user interface (GUI). A Windows-based application thus starts and then waits until the user clicks a button or selects a menu item before anything happens. This is known as event-driven programming.

Event Driven Architecture Topologies

- ❖ The event-driven architecture pattern consists of two main topologies, the mediator and the broker.
- ❖ The mediator topology is commonly used when you need to orchestrate multiple steps within an event through a central mediator, whereas the broker topology is used when you want to chain events together without the use of a central mediator.
- ❖ Technologies that help Mediator topology: WebSphere Process Server, jBPM, Mule ESB and OpenESB
- ❖ Technologies that help Broker topology: WebSphereMQ, RabbitMQ, HornetQ, Kafka



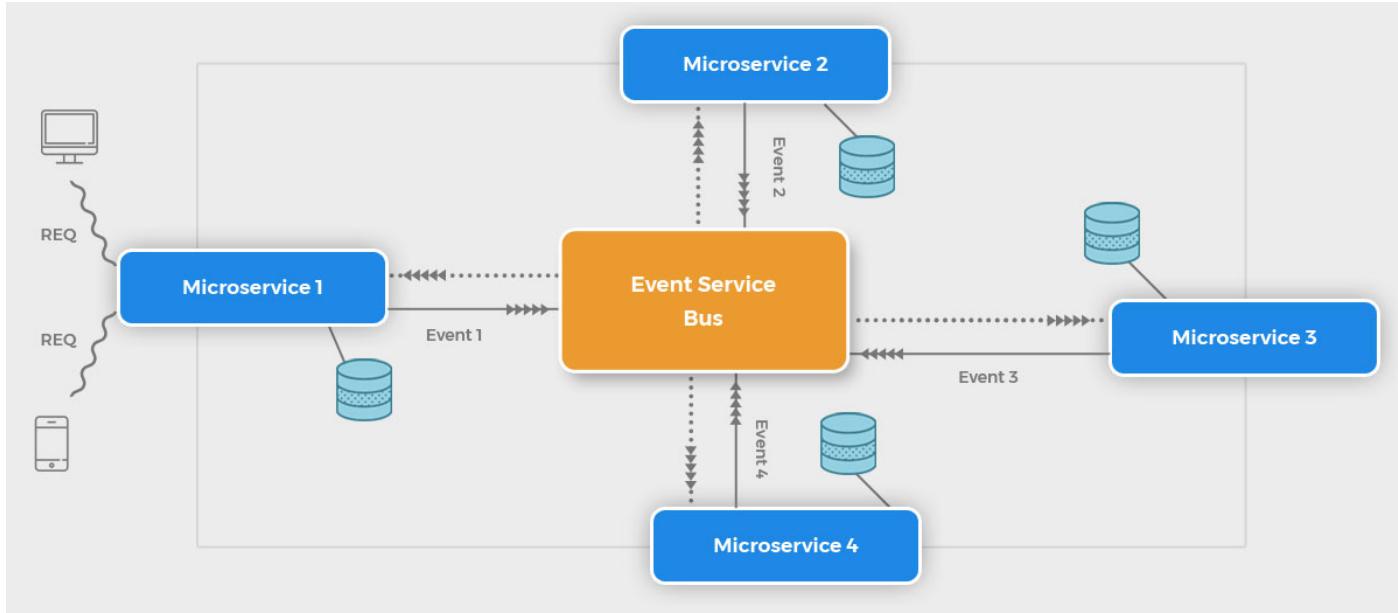
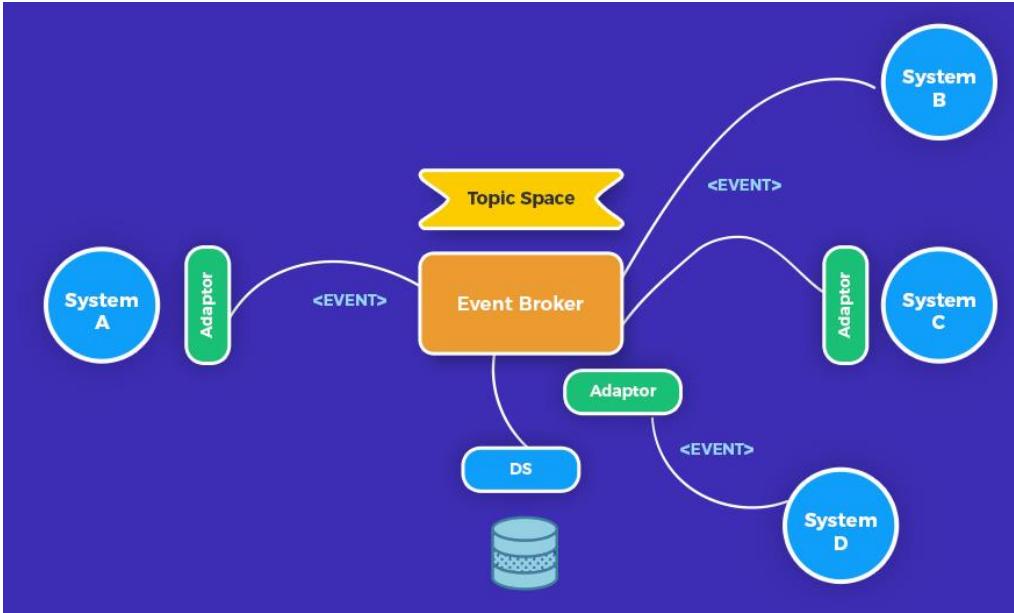
Stock Trade Example:

- A single event to place a stock trade might require you to first validate the trade, then check the compliance of that stock trade against various rules, assign the trade to a broker, calculate the commission, and finally place the trade with that broker.
- All of these steps would require some level of orchestration to determine the order of steps and which can be done serially or in parallel.

Ecommerce example:

- Product sales data from online ecommerce applications sales and in-store POS sales data needed by accounts department as well as Inventor/warehouse management department
- Offers and Discounts data decided by online marketing depart are needed by Stores
- Sales data needed by Delivery team and warehouse team for order package/dispatch

Typical use case of Event Broker



Ecommerce example (Publish –subscribe or Hub-Spoke)

- Product sales data from online ecommerce applications sales and in-store POS sales data needed by accounts department as well as Inventor/warehouse management department
- Offers and Discounts data decided by online marketing depart are needed by Stores
- Sales data needed by Delivery team and warehouse team for order package/dispatch

Web Sphere MQ is used as Message Broker for one of the Chilean Retail giant WebSphere Ecommerce Server (adapter)

JDA was used for Inventory Management (Adapter)

Customer services and data are on IBM Main frame with DB2 (custom adapter)

EDA with Microservices approach

Pipe and Filter Architecture pattern

Pipe and Filter is a simple architectural pattern that connects a number of components that process a stream of data, each connected to the next component in the processing pipeline via a **Pipe**.

The Pipe and Filter architecture is inspired by the Unix technique of connecting the output of an application to the input of another via pipes on the shell.

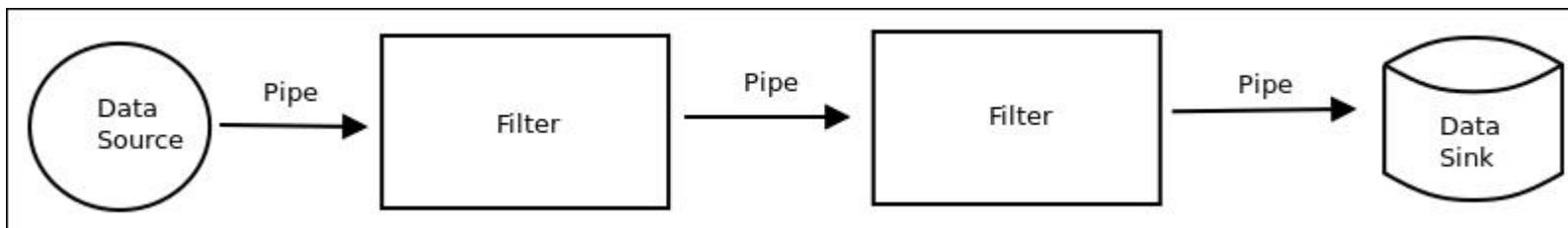
```
$> ls -al | pg
```

(Paginated list of files)

```
$> find . -type f -exec ls -s {} \; | sort -n | head -5
```

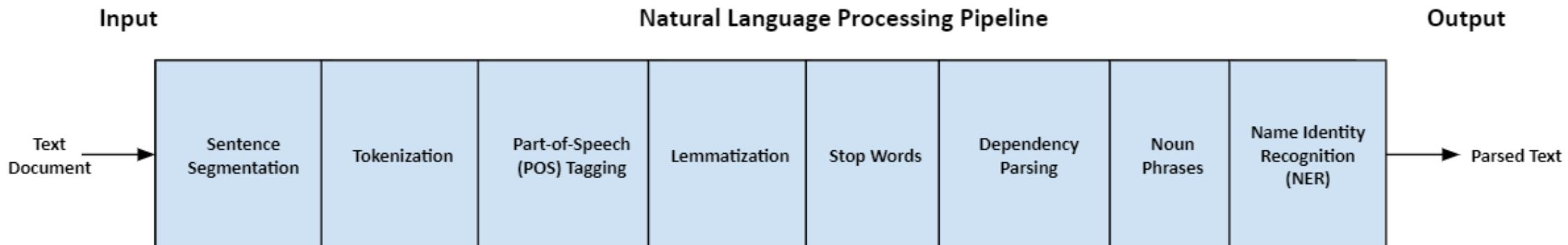
(Finding top 5 big files in the current directory)

The pipe and filter architecture consists of one or more data sources. The data source is connected to data filters via pipes. Filters process the data they receive, passing them to other filters in the pipeline. The final data is received at a **Data Sink**:



Pipe and Filter examples

❖ Natural Language Processing pipeline

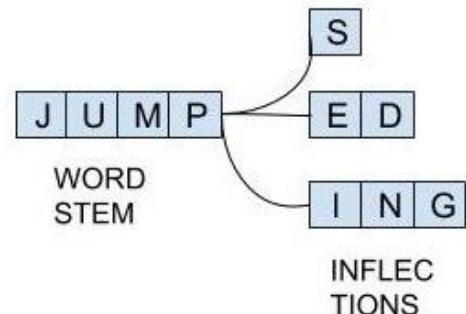


Sentence Segmentation

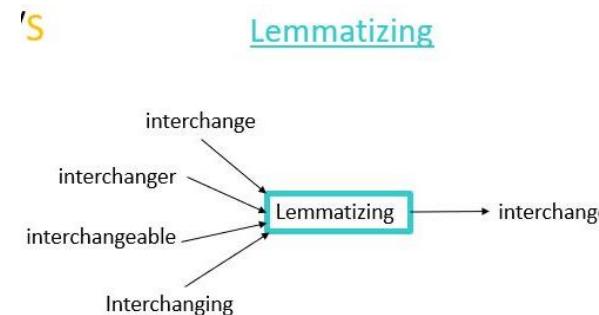
Hello world. This blog post is about sentence segmentation. It is not always easy to determine the end of a sentence. One difficulty of segmentation is periods that do not mark the end of a sentence. An ex. is abbreviations.



- Hello world.
- This blog post is about sentence segmentation.
- It is not always easy to determine the end of a sentence.
- One difficulty of segmentation is periods that do not mark the end of a sentence.
- An ex. is abbreviations.

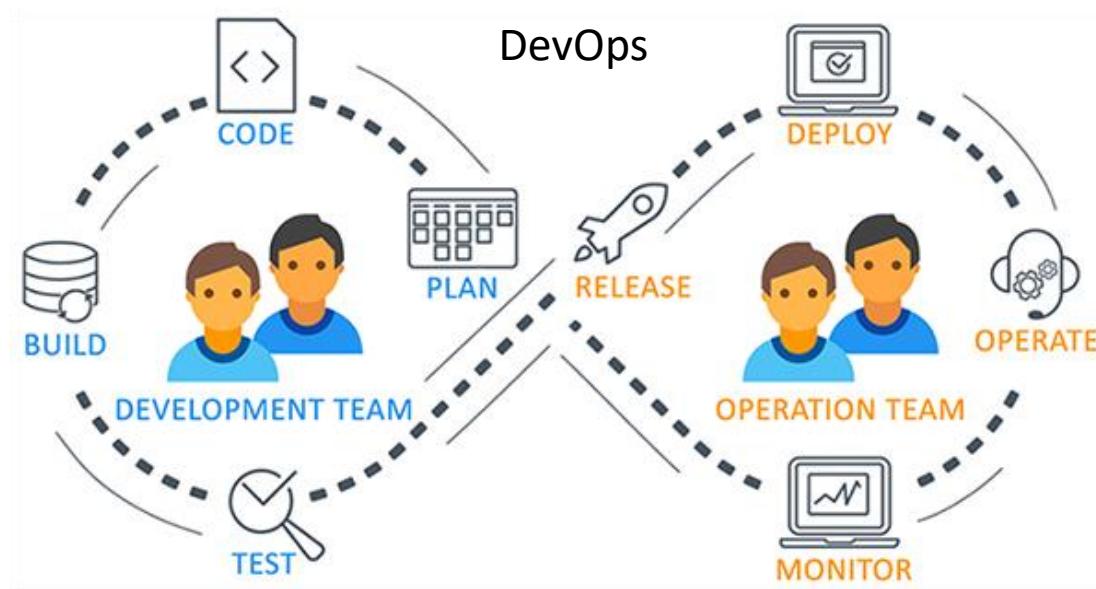
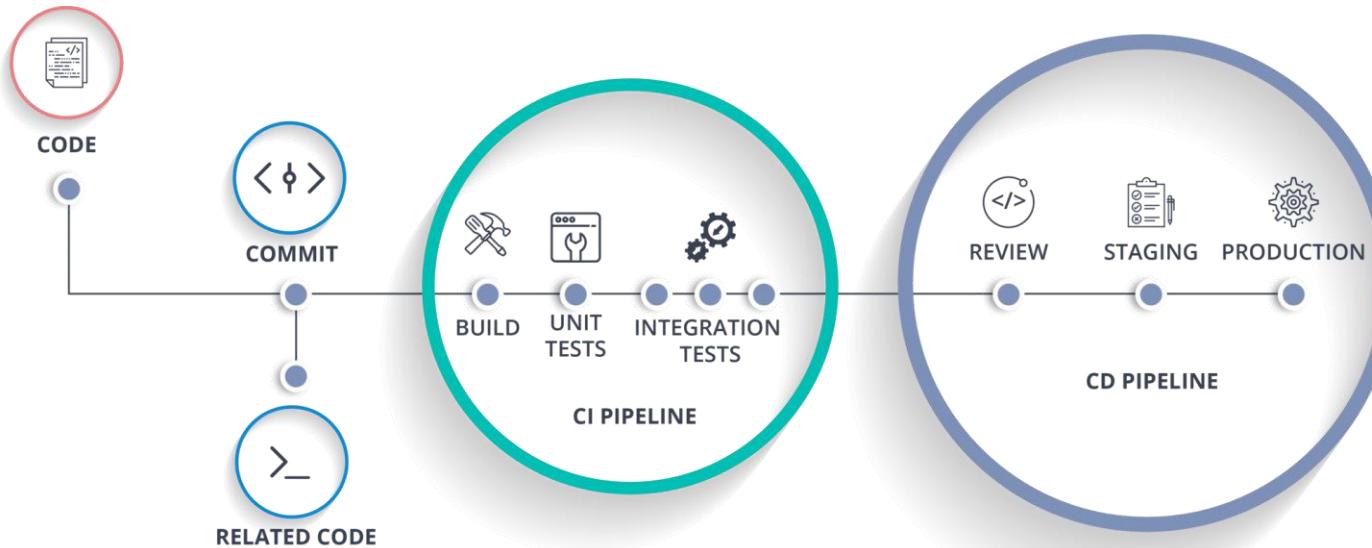


Word stem and its inflections (Source: Text Analytics with Python, Apress/Springer 2016)



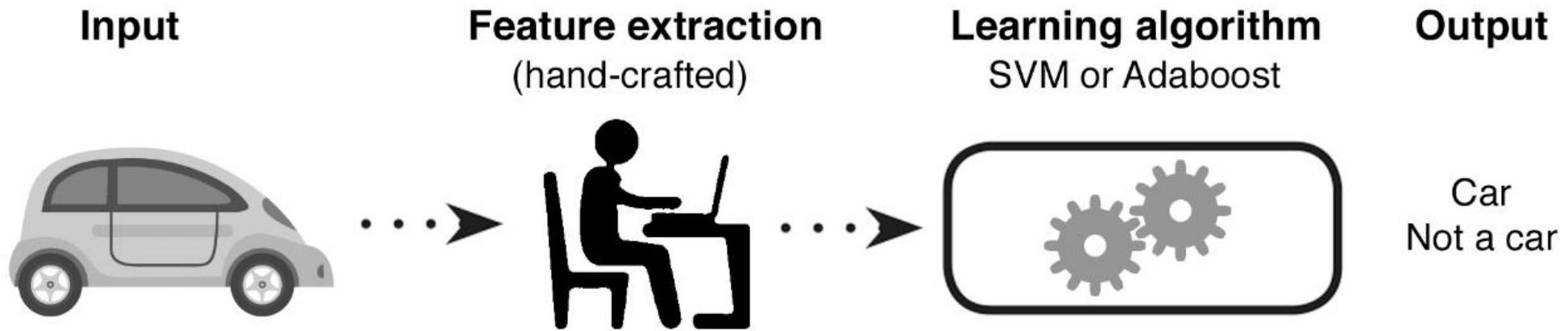
Label	Token	POS	Category	nsubj	aux	ROOT	det	dobj	p
John	can	hit	the	ball
NNP	MD	VB	DT	NN
N	MD	V	DT	N

DevOps Continuous Integration and Continuous Delivery/Deployment Pipeline

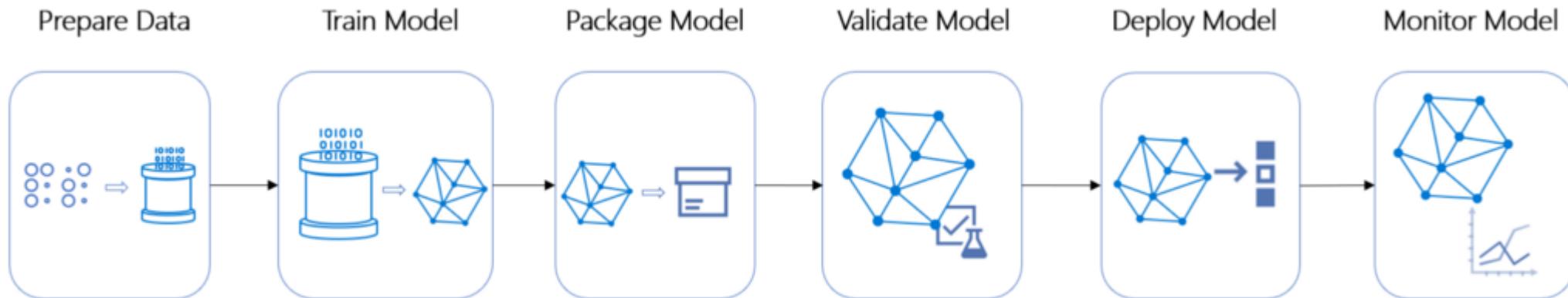


Computer Vision and Machine Learning pipeline examples

Computer Vision Pipeline for a simple classification problem



Typical Machine Learning Pipeline:



Master Slave Pattern

- ❖ The Master–Slave architectural pattern is used to improve system reliability, efficiency and performance by dividing work between the master and slave components. Each component has distinct responsibilities. All slave components have identical or at least similar work, and that work must be defined prior to runtime.
- ❖ The Master-Slave pattern is often used for multi-threaded applications in which many instances of the same problem must be solved. The master creates and launches slaves to solve these instances in "parallel". When all of the slaves have finished, the master harvests the results.
- ❖ Master-Slave pattern is also used for user interfaces and servers. In both cases the master listens for commands coming either from the user or from clients. When a command is received, a slave is launched to execute the command while the master resumes listening for more commands (e.g., suspend last command)

Usage

- ❖ In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
- ❖ Multiple UAVs coordinate with each other to complete a task. They communicate with each other wirelessly and take decisions. For UAV swarms coordination, master-slave communication approach is used.
- ❖ Peripherals connected to a bus in a computer system/microcontroller using I2C (sensors, LCD display, serial EEPROM, ADC) and SPI protocols (sensors, LCD display , SD card, serial EEPROM).

Blackboard Pattern

- ❖ Problem: Flexible integration of partial solutions
- ❖ Solution: The idea of blackboard architecture is similar to the blackboard setting used to solve problems. The system consists of 1 Blackboard, Many knowledge sources and 1 control

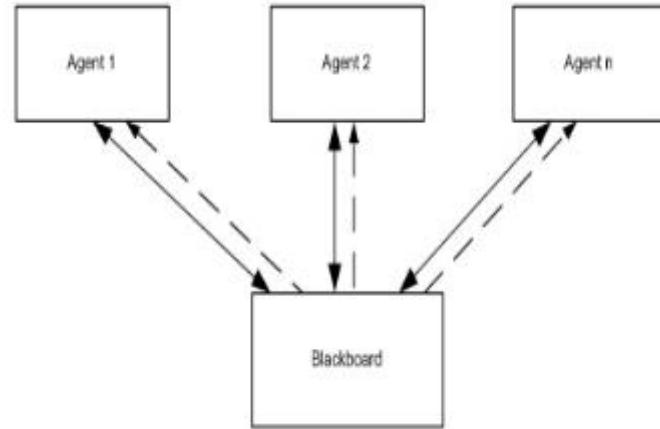
Blackboard is common data structure. Knowledge sources independently fill/modify the blackboard contents. Control monitors changes and launches next knowledge sources

❖ Blackboard Essentials

- Blackboard allows multiple processes (or agents) to communicate by reading and writing requests and information to a global datastore.
- Each participant agent has expertise in its own field, and has a kind of problem solving knowledge (knowledge source) that is applicable to a part of the problem, i.e., the problem cannot be solved by an individual agent only.
- Agents communicate strictly through a common blackboard whose contents is visible to all agents. When a partial problem is to be solved, candidate agents that can possibly handle it are listed.
- A control unit is responsible for selecting among the candidate agents to assign the task to one of them and overall supervision control.

❖ Examples:

- DeepQA Jeopardy Challenge
- Global Logger for multiple processes (using shared memory on Unix)
- Speech Transcription Engine
- NLP Tasks such as Named Entity Recognition, Topic



Guidelines to Design a Blackboard:

- (1) Define the problem
- (2) Define the solution space for the problem
- (3) Divide the solution process into steps
- (4) Divide the knowledge into specialized knowledge sources with certain tasks
- (5) Define the vocabulary of the blackboard
- (6) Specify the control of the system
- (7) Implement the knowledge sources

Advantages:

- (1) Easy to add/update new/existing knowledge source
- (2) Knowledge source agents can work in parallel as work independently
- (3) Reusability of Knowledge source agents

Disadvantages:

- (1) Tight dependency between knowledge source and blackboard
- (2) Synchronization of multiple agents is an issue
- (3) System Debugging and testing is difficult.

Deep QA system for Jeopardy challenge

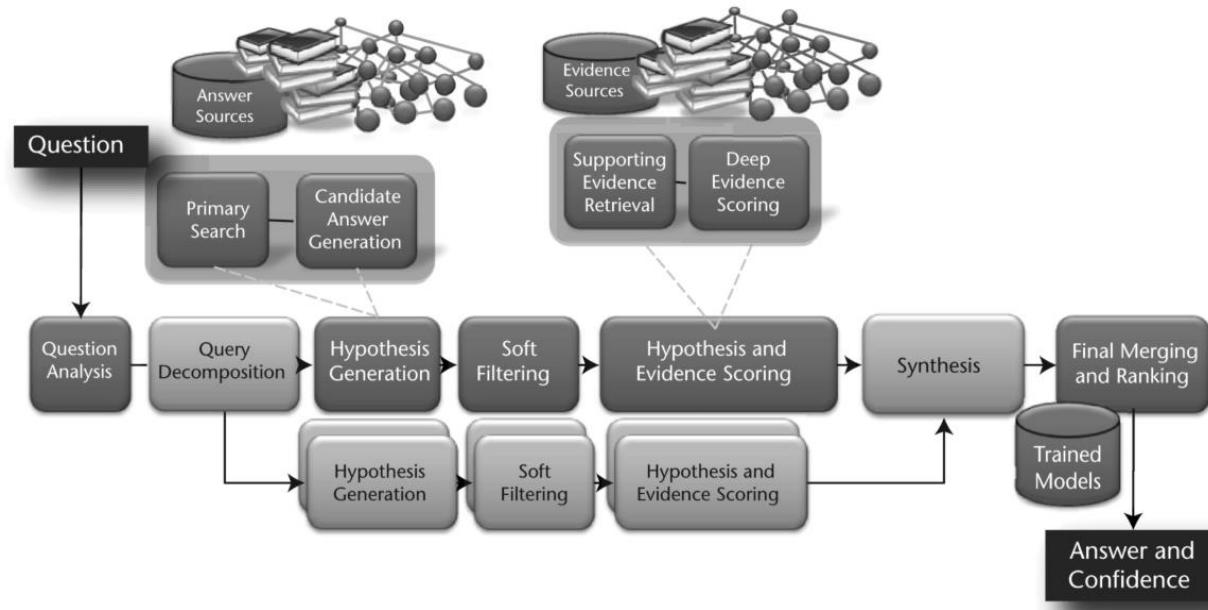
IBM Research undertook a challenge to build a computer system (a real-time automatic contestant) that could compete at the human champion level in real time on the American TV quiz show, Jeopardy.

The Jeopardy Challenge helped address requirements that led to the design of the DeepQA architecture and the implementation of Watson. After three years of intense R&D by 20 researchers, Watson performed at human expert levels in terms of precision, confidence, and speed at the Jeopardy quiz show in 2011.

The *Jeopardy* Challenge

- ❖ Meeting the *Jeopardy* Challenge requires advancing and incorporating a variety of QA technologies including **parsing, question classification, question decomposition, automatic source acquisition and evaluation, entity and relation detection, logical form generation, and knowledge representation and reasoning**.
- ❖ Winning at *Jeopardy* requires accurately computing confidence in your answers. The questions and content are ambiguous and noisy and none of the individual algorithms are perfect.
- ❖ Therefore, each component must produce a confidence in its output, and individual component confidences must be combined to compute the overall confidence of the final answer.
- ❖ The final confidence is used to determine whether the computer system should risk choosing to answer at all. In *Jeopardy* parlance, this confidence is used to determine whether the computer will “ring in” or “buzz in” for a question.
- ❖ The confidence must be computed during the time the question is read and before the opportunity to buzz in. This is roughly between 1 and 6 seconds with an average around 3 seconds.

A derivative of the Unstructured Information Management Architecture (UIMA), used in the IBM Watson project, which combines a pipeline of modules with an unstructured blackboard.



Category: General Science

Clue: When hit by electrons, a phosphor gives off electromagnetic energy in this form.

Answer: Light (or Photons)

Category: Lincoln Blogs

Clue: Secretary Chase just submitted this to me for the third time; guess what, pal. This time I'm accepting it.

Answer: his resignation

Category: Head North

Clue: They're the two states you could be reentering if you're crossing Florida's northern border.

Answer: Georgia and Alabama

References

- ❖ <https://martinfowler.com/eaaCatalog/index.html>
- ❖ <https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013>
- ❖ Buschmann F., Meunier R., Rohnert H., SommerladP. & Stal M. (1996). Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons.
- ❖ <https://xavierantony.wordpress.com/2010/08/24/windows-messaging-architecture/>
- ❖ <https://aws.amazon.com/lambda/>
- ❖ <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
- ❖ <https://microservices.io/patterns/microservices.html>

Thank You



Martin Fowler

fowler@acm.org

Refactoring: Doing Design After the Program Runs

If you have worked with objects for any length of time, you should have heard about the notion of iterative development. The idea of iteration is that you cannot get a design right the first time, you need to refine it as you build the software. By iterating over the design several times, you get a better design. Even throwing away code helps; indeed it is the sign of a good project that it does regularly throw away code. To not do so indicates a lack of learning—and keeping bad ideas around.

Iteration is a great principle to discuss, but it has some problems in practice. If you make a change to existing code, does that not carry the risk of introducing bugs? While people have come up with various techniques and methods for design in advance, there is not much discussion of how to apply them in an iterative process, or how to do the iteration in a controlled and efficient manner.

Refactoring is the first technique I've come across that is explicitly about doing iterative development in a controlled manner. It starts with software that currently works but is not well suited to an enhancement you wish to make. Refactoring is the controlled process of altering the existing software so its design is the way you want it now, rather than the way you wanted it then. It does this by applying a series of particular code transformations, each of which are called *refactorings*. Each refactoring helps change the code in a way that both is rapid and does not introduce bugs. (Yes, I know that means the word refactoring means two different things—I guess overloading is just ingrained in this industry!)

How does refactoring do this magic? Essentially, there are two routes. The first is manual; the second relies on tools. Although using tools is less common at the moment, I'll start with that.

Refactoring With Tools The essence of the tools-based approach is the notion of the *semantics-preserving transformation*. This is a transformation that you can prove will not change the execution of the program. An example of this is a refactoring called *extract method*. If you have a long section of procedural code, you can make it easier to read by taking a suitable chunk of the procedure and turning it into a separate method. To do this with a tool, you select the code you wish to extract, and the tool analyzes this code, looking for temporary variables and parameters. Depending on what the selected code does with these

local variables, you may have to pass in parameters, return a value, or even not be able to do the extraction. The tool does this analysis, prompts for the new method name, asks you to name any parameters, and creates the new method with a call from the old one. The program works exactly the same as it did before but is now a little easier for a human to read. This is important: Any damn fool can write code that a computer can understand, the trick is to write code that humans can understand.

The tool builder has to prove that the refactoring is semantics preserving, then figure out how to provably carry out the transformation in the code. It's hairy stuff and requires a lot of knowledge of compiler technology, but it can be done. The benefit is immediate. Once the transformation is encoded, you can use it with complete confidence. A semantics-preserving transformation is never going to add a bug to your program.

Tools like this are not science fiction. Bill Opdyke, while a student at the University of Illinois, proved several of these refactorings. Since then, two other graduate students, John Brant and Don Roberts, have produced a refactoring browser for Smalltalk that implements these refactorings, and a few more. If you are a Smalltalker, you should download it from www.cs.uiuc.edu/users/droberts/Refactory.html.

The refactoring browser is an awesome tool. With it I can safely reorganize some pretty ugly code (and yes even Smalltalk can get ugly). But what if you are working outside of Smalltalk? Is refactoring still applicable? The answer is yes, although not with tool support.

Refactoring Without Tools Although manual refactoring is not as easy, it is still possible—and useful. It boils down to two principles: take small steps and test frequently.

Humans can use the same semantics-preserving transformation tools use, but in method extraction you have to look at the local variables yourself. It takes a little longer, but it isn't too difficult, because you're looking at only a small section of code. You then move the code over, put in the call, and recompile.

If you did it correctly, you won't get any bugs. Of course, that's a big if, so this is where tests come in. With manual refactoring you need to have a battery of tests that exercise the code sufficiently to give you confidence that you won't introduce new bugs. If you build self-testing code, then you will already have those tests. If not, you have to build them yourself, but of course tests are useful anyway, since they make it easier to add new function as well as refactor.

Martin Fowler is an independent consultant based in Boston, MA.

56

MindQ Ad

Refactoring to Understand Code When you follow a rhythm of small change, test, small change, test, you can make some remarkably large changes to a design. I've gone into some pretty nasty lumps of code, and after a few hours found class structures that radically improve the software design. I don't usually have the design in mind when I start. I just go into the code and refactor initially just to understand how the code works. Gradually, as I simplify the code, I begin to see what a better design might be and alter the code in that direction.

Refactoring to understand code is an important part of the activity. This is obviously true if you are working with someone else's code, but it is often true with your own code as well. I've often gone back to my own programming and not fully understood what I was doing from the code, or gained a better understanding by comparing it to later work. Of course, you can do a similar thing by commenting, but I've found it better to try to refactor the code so its intention is clear from the code, and it does not need so many comments. Some refactors go so far to claim that most comments are thus rendered unnecessary. I don't go that far, but I do prefer to reach clarity through good factoring if I can.

The Value of Refactoring Refactoring has become a central part of my development process, and of the process I teach through my seminars and consulting work. It's a design technique that is a great complement to the up-front design techniques advocated by the UML and various methods.

Its great strength is that it works on existing software. I'm rarely faced with a green field. Often I have to work with some form of existing code base. Refactoring provides a way to manipulate and improve the code base without getting trapped in bugs.

When working with new code, it gives developers the ability to change designs and make the iterative development process much more controlled. I've noticed that it makes a significant improvement to development speed. When trying to add new function to software, several projects have seen how it is quicker to first refactor the existing software to make the change easier. Thus I don't recommend setting aside time to refactoring. It should be something you do because you need to add a feature or fix a bug. That also makes it easier to justify to skeptical managers.

The biggest problem with refactoring at the moment is finding out more about how to do it. I'm in the process of writing about what I, and various others, have learned about the process, but that is still a way from publication. You can find some more information, including references and an introductory example of refactoring, at ourworld.compuserve.com/homepages/Martin_Fowler/.

I hope I've stimulated you to find out more about refactoring. I believe it is going to be one of crucial new developments of the next few years, and tools that include refactorings will be very important to software developers. I'd be interested to know what you think; drop me a note at fowler@acm.org. ☺

Patterns

Martin Fowler

I should come as no surprise that I'm a big fan of software patterns. After all, I just finished my second book on the subject, and you don't put forth that kind of effort without believing in what you're doing. Working on the book, however, reminded me of many things about software patterns that are not fully understood, so this seemed like a good time to talk about them.

The biggest software patterns community is rooted in the object-oriented world. This community includes the people who wrote the classic Gang of Four book (E. Gamma et al., *Design Patterns*, Addison-Wesley, 1995) and the Hillside group that organized the worldwide PloP (Pattern Languages of Programs) conferences (see <http://hillside.net>). Most patterns books have come out of this community—but not all. There are good books, such as David Hay's *Data Model Patterns* (Dorset House, 1996), written by people who've had little or no contact with this group.

Even with the Hillside group, there's a lot of disagreement about what's important about patterns—and that mix of views grows even larger with those who don't get involved in the Hillside group's efforts. So there are many opinions on what makes a pattern important. Following are my views.

Why patterns interest me

Patterns provide a mechanism for rendering design advice in a reference format. Software design is a massive topic, and when faced with a design problem, you must be

able to focus on something as close to the problem as you can get. It's frustrating to find an explanation of what I need to do buried in a big example that contains 20 things that I don't care about but must understand to complete the thing I do care about.

So for me an important problem is how to talk about design issues in a relatively encapsulated way. Of course it's impossible to ignore all the inter-relationships, but I prefer to minimize them. Patterns can help by trying to identify common solutions to recurring problems. The solution is really what the pattern is, yet the problem is a vital part of the context. You can't really understand the pattern without understanding the problem, and the problem is essential to helping people find a pattern when they need it.

Similarly, abstract discussions of principles—which I often write about in this column—are important, but people need help applying these principles to more concrete problems.

When people write patterns, they typically write them in some standardized format—as befits a reference. However, there's no agreement as to what sections are needed because every author has his or her own ideas. For me, there are two vital components: the how and the when. The how part is obvious—how do you implement the pattern? The when part often gets lost. One of the most useful things I do when trying to understanding a pattern, one I'm either writing or reading, is ask, "When would I *not* use this pattern?" Design is all about choices and tradeoffs; consequently, there usually isn't one design approach that's always the right one to use. Any pattern should discuss alter-



natives and when to use them rather than the pattern you're considering.

One of the hardest things to explain is the relationship between patterns and examples. Most pattern descriptions include sample code examples, but it's important to realize that the code examples aren't the pattern. People considering building patterns tools often miss this. Patterns are more than macro expansions—every time you see a pattern used, it looks a little different. I often say that patterns are half-baked—meaning you always have to finish them yourself and adapt them to your own environment. Indeed, implementing a pattern yourself for a particular purpose is one of the best ways to learn about it.

Patterns and libraries

This raises an interesting point—if a pattern is a common solution, why not just embed it into libraries or a language? Then people wouldn't need to know the pattern. Certainly, embedding patterns into libraries is common—indeed, usually, it's the other way around. Pattern authors see many libraries doing a similar thing and consequently identify the pattern. This provides value in several ways.

First of all, it can help people understand how the library feature works to extract the library's specific context. A library typically combines many things at once, so again you run into the problem of having to understand a dozen things. A well-written set of patterns can help explain these concepts.

Second, people often move between programming environments, so they might be familiar with a particular solution but not how to implement it in a new environment. Understanding the underlying pattern behind library features helps a great deal in making this connection.

Finally, even if a library implements a pattern, you must still decide how to use it. You might also need to know more about what implementation strategies the library uses and whether they are appropriate for a particular problem. A library can

only implement the “how” part of a pattern—you still have to answer the “when.” In this case, the presence of a library implementation alters the tradeoff. If a library implements a solution that isn't ideal, you can still choose to use it rather than implement the ideal one yourself.

Patterns and the expert

One of the hardest things about patterns is that they are, by definition, nothing new. If you've been working in a field for a while and have become very skilled in it, then a patterns book in that field shouldn't teach you anything new. Are patterns at all valuable to experts? They obviously offer less value to experts than to someone coming to grips with the field, but there is still something to be gained by looking at patterns that merely capture what you already know.

The primary value is that they can help you teach those around you. For me, the driving force behind writing comes from seeing that there's knowledge to be shared. An expert in a team can use written patterns to help educate other team members. The expert can help the team review the general case, which will come with simplified and encapsulated examples, but more importantly, he or she can then show the team how the patterns should or shouldn't be used in the project at hand. So if you're an expert in your field, you might rate the quality of a patterns book based

on how it helps you teach your colleagues rather than on what you learned from it.

The other value of patterns to experts is as a standard vocabulary. In the OO world, we can talk about singletons, strategies, decorators, and proxies, confident that a moderately experienced designer will have a good chance of understanding what we mean without a lot of extra explanation. This vocabulary makes it easier to discuss our designs.

Overuse

One of the things that can be a problem is that people can think that patterns are unreservedly good and that a test of a program's quality is how many patterns are in it. This leads to the apocryphal story of a hello world application with three decorators, two strategies, and a singleton in a pear tree. Patterns are not good or bad—rather, they're either appropriate or not for some situations. I don't think it's wrong to experiment with using a pattern when you're unsure, but you should be prepared to rip it out if it doesn't contribute enough.

I don't subscribe to the opinion that there are few remaining patterns to gather. In both of my patterns books, I felt I did little more than scratch the surface. There is more room for people to look at the systems that have been built and try to identify and describe the patterns involved. This strikes me as an ideal task for academia, although, sadly, the fact that patterns are by definition nothing new seems to make that impossible.

As a field we have much to learn, which is why software development is so much fun. But I think it's frustrating when we don't take the time to learn from our own efforts. ♦

Patterns are half-baked—meaning you always have to finish them yourself and adapt them to your own environment.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.

Software Architecture

Y. Raghu Reddy

Software Engineering Research Centre

Traditional Engineering

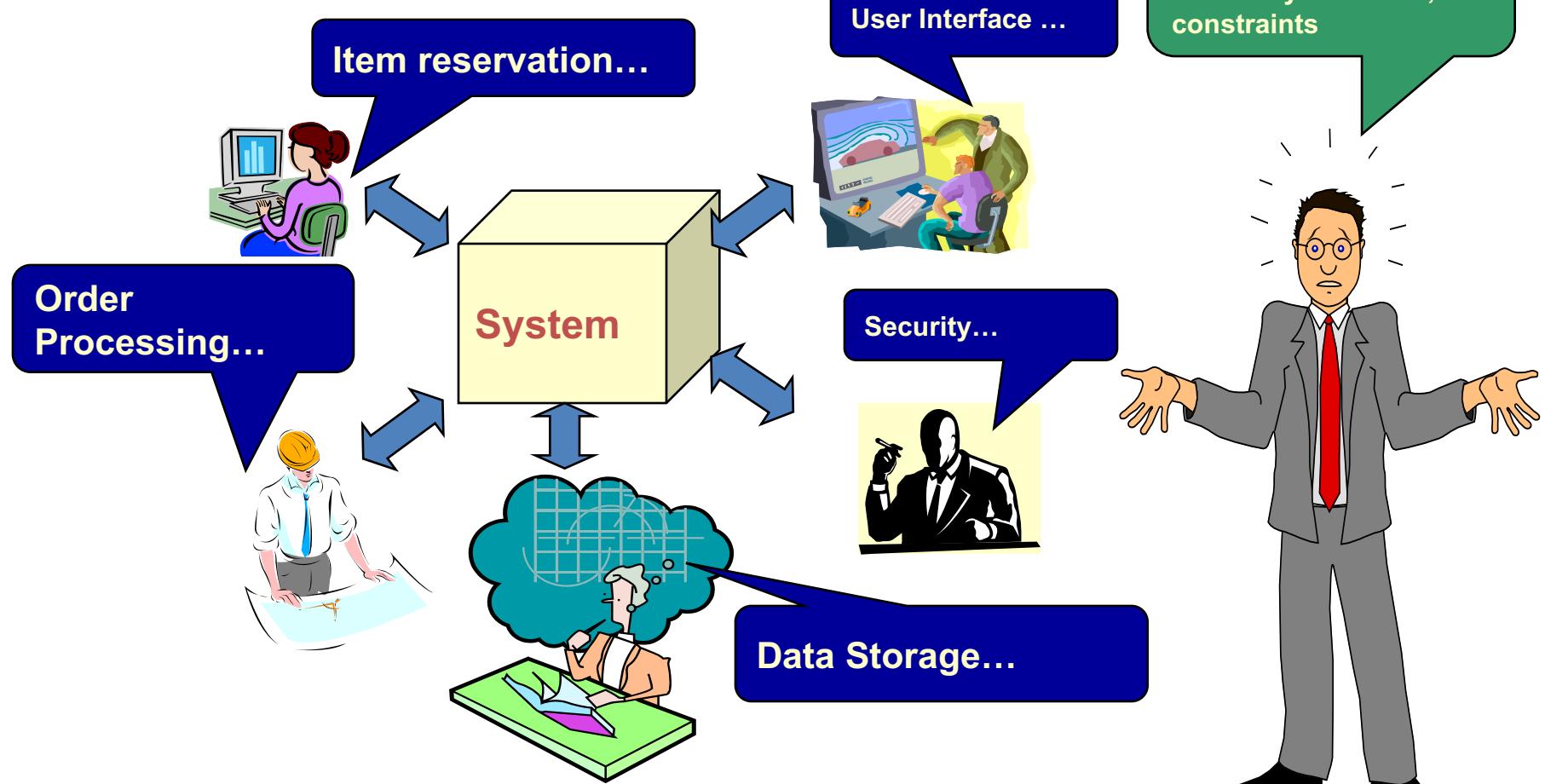
- Traditional Engineering has to abide by the laws of physics.
- Material artifacts are not malleable
- Chances of do over are less
 - Can't build a bridge and then check if it breaks or if everything is not aligned properly !!!



Can Software Engineering mimic Traditional Engineering ?

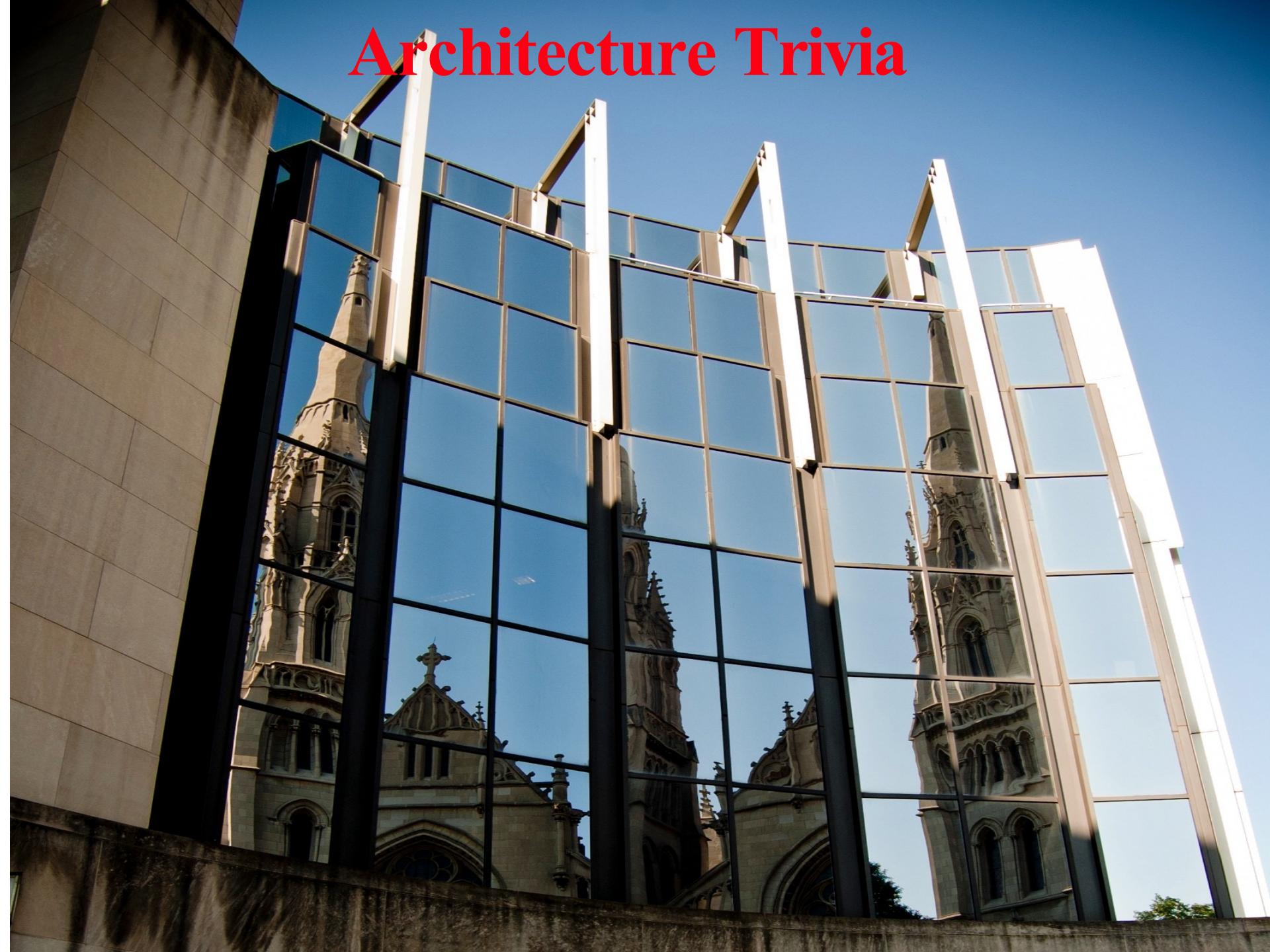
- Software Engineering deals with abstract artefacts (non-real),
- Software artifacts are highly malleable
- We can make, break and make it again
 - as long as they don't get complex enough !!!

Software Complexity



- Are functional and quality requirements the same?
- Are they Orthogonal?

Architecture Trivia



Agenda items next two weeks

- Importance of Architecture
- Architecture Descriptions
- Architecture Practices
 - Patterns
 - Anti-patterns
- Analysis of Architecture
- Case Studies

When did “Architecture” get started?



Imhotep - Pyramid of Djoser (step Pyramid) at Saqqara in Egypt in 2630 – 2611 BC

What about Software Architecture?

- Introduced in the Late 60s, Early 70s... by Djikshtra & David Parnas
- Many things to many people
 - Term has be used, misused, abused...
- Software Architecture has to do with two things (in very simple terms):
 - Doing the right things
 - Doing the things right

Software Architecture Definitions ... 1 / 2

- Vehicle for stakeholder communication
- Depiction of system
- Serves as a Blueprint
- Primary carrier of system qualities
- An early analysis of a design approach
- Architectural Model = Top level structure + organizing principles
 - Top level structure: partitioning system into high level components (usually resulting in modules)
 - Organizing principles: a few concepts and design decisions which set the course of implementation

Software Architecture Definitions ... (2/2)

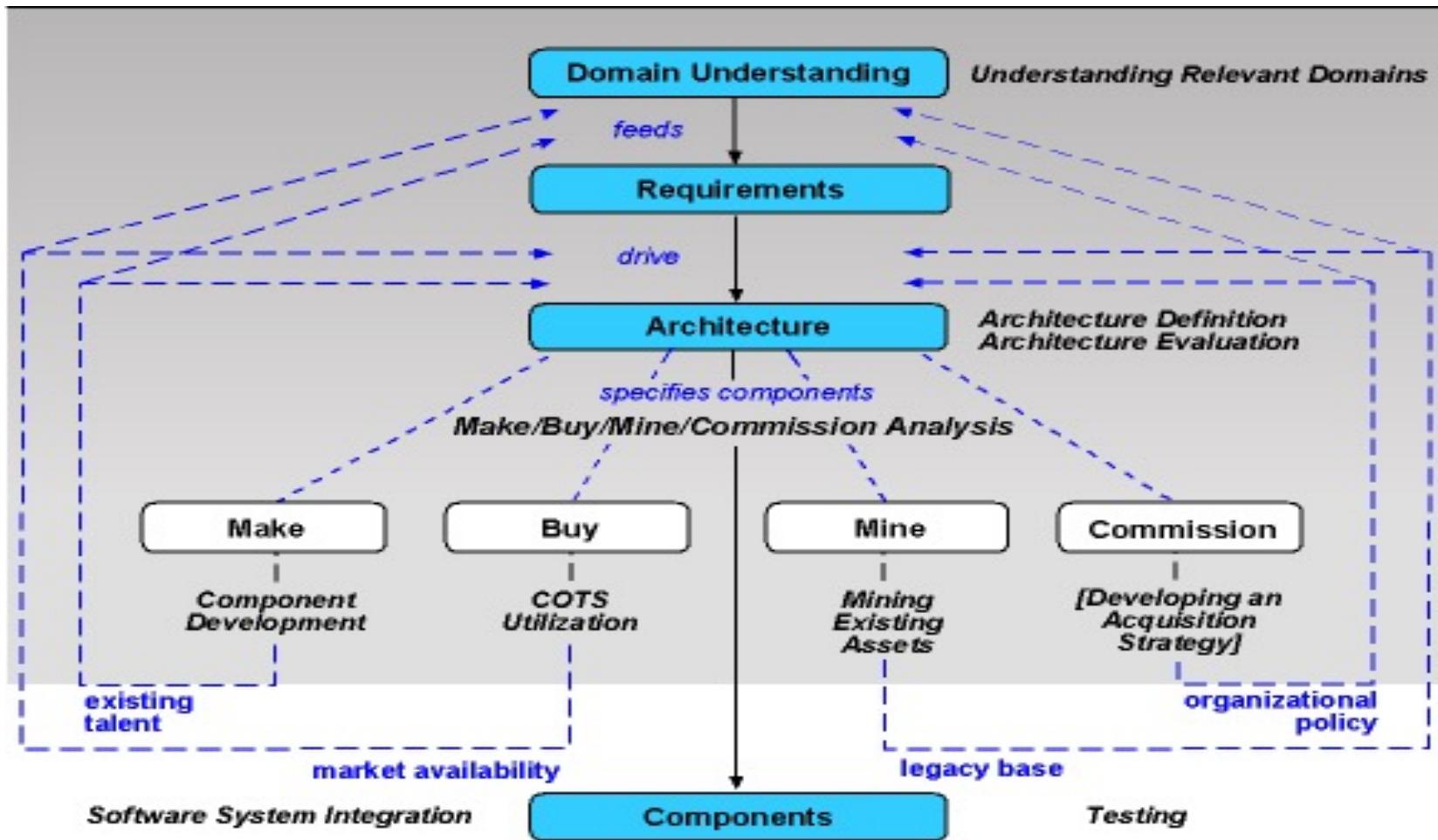
“ The software architecture of a program or computing system is the **structure** or **structures** of the system, which comprise **software elements**, the **externally visible properties** of those elements, and the **relationships** between them.”

- Len Bass et al.

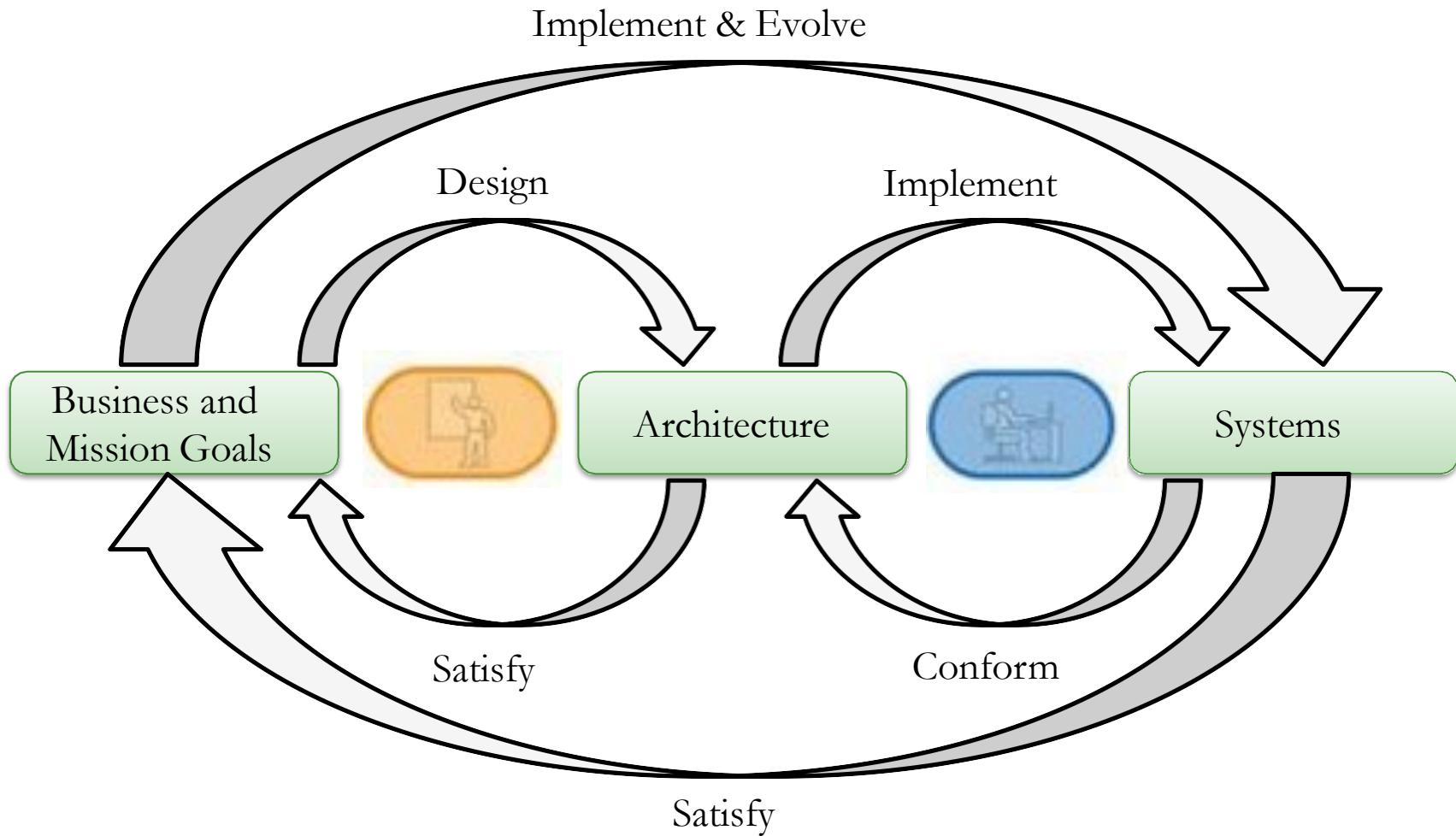
Importance of Software Architecture

- Tool for Communication
 - Different types of diagrams can be used to represent different viewpoints (from a stakeholders perspective)
- Facilitates decision making early in the life cycle
 - Should we use natural lighting in a building?
 - Should be use LEDs? Should be use Philips LEDs? How many watts?
 - Should we use a web based client?
 - What protocol should the web service use for communication?
- Architecture as a transferable, re-usable model
 - Artifacts are transferrable across systems (product lines)
 - Reasonable (constrained) set of alternatives

Architecture in SE practices



Essentially, Meeting Stakeholder Needs



Converting Stakeholder requirements to Architectural representation

- Documenting in English
 - Template based
- Architectural Description Languages (ADL)
- Description techniques/Frameworks
 - 4+1 view, TOGAF, Zachman Framework, DoDAF, etc

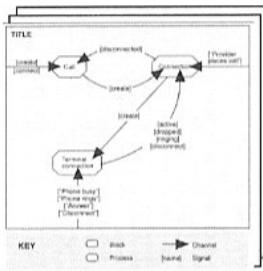
View Documentation Template (in English)

1. Primary presentation – elements and their relationships
2. Element catalog -- explains the picture
3. Context diagram -- how the system relates to its environment
4. Variability guide – how to exercise any variation points
5. Architecture background – why the design reflected in the view came to be
6. Glossary of terms used
7. Other information

Pictorially:

Views

Section 1. Primary Presentation of the View



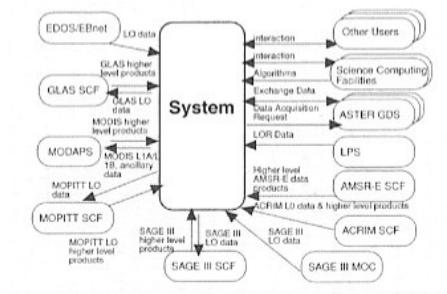
OR

Textual version
of the primary
presentation

Section 2. Element Catalog

- Section 2.A Elements and their properties
- Section 2.B Relations and their properties
- Section 2.C Element interfaces
- Section 2.D Element behavior

Section 3. Context Diagram



Section 4. Variability Guide

Section 5. Architecture Background

- Section 5.A Design rationale
- Section 5.B Analysis of results
- Section 5.C Assumptions

Section 6. Glossary of Terms

Section 7. Other Information

Documenting Interfaces

1. Interface identity - unique name
2. Resources provided
3. Locally defined data types - if used
4. Exception definitions - including handling
5. Variability provided - for product lines
6. Quality attribute characteristics - what is provided?
7. Element requirements
8. Rationale and design issues - why these choices
9. Usage guide - protocols

Architecture Description Languages (ADLs)

- Provide a formal way of representing architecture
- Intended to be human and machine readable
- Support describing a system at a higher level than previously possible
- Permit analysis of architectures – completeness, consistency, ambiguity, and performance
- Can support automatic generation of software systems

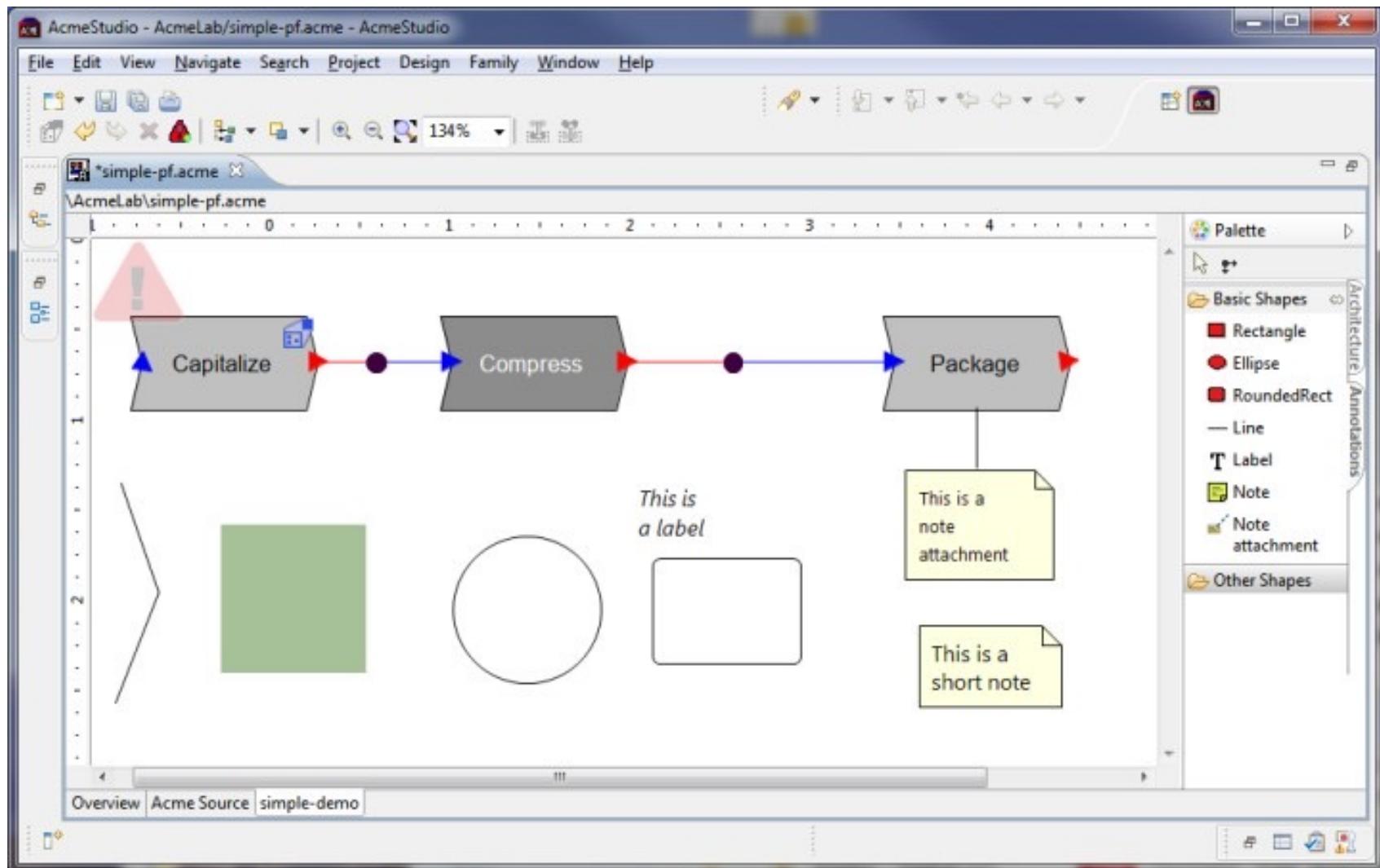
ADLs - Negatives

- No universal agreement on what ADLs should represent, particularly as regards the behavior of the architecture
- Representations currently in use are relatively difficult to parse and are not supported by commercial tools
- Most ADL work today has been undertaken with academic rather than commercial goals in mind

Candidate ADLs

- Leading candidates
 - ACME (CMU/USC)
 - Rapide (Stanford)
 - Wright (CMU)
 - Unicon (CMU)
- Secondary candidates
 - Aesop (CMU)
 - MetaH (Honeywell)
 - C2 SADL (UCI)
 - SADL (SRI)

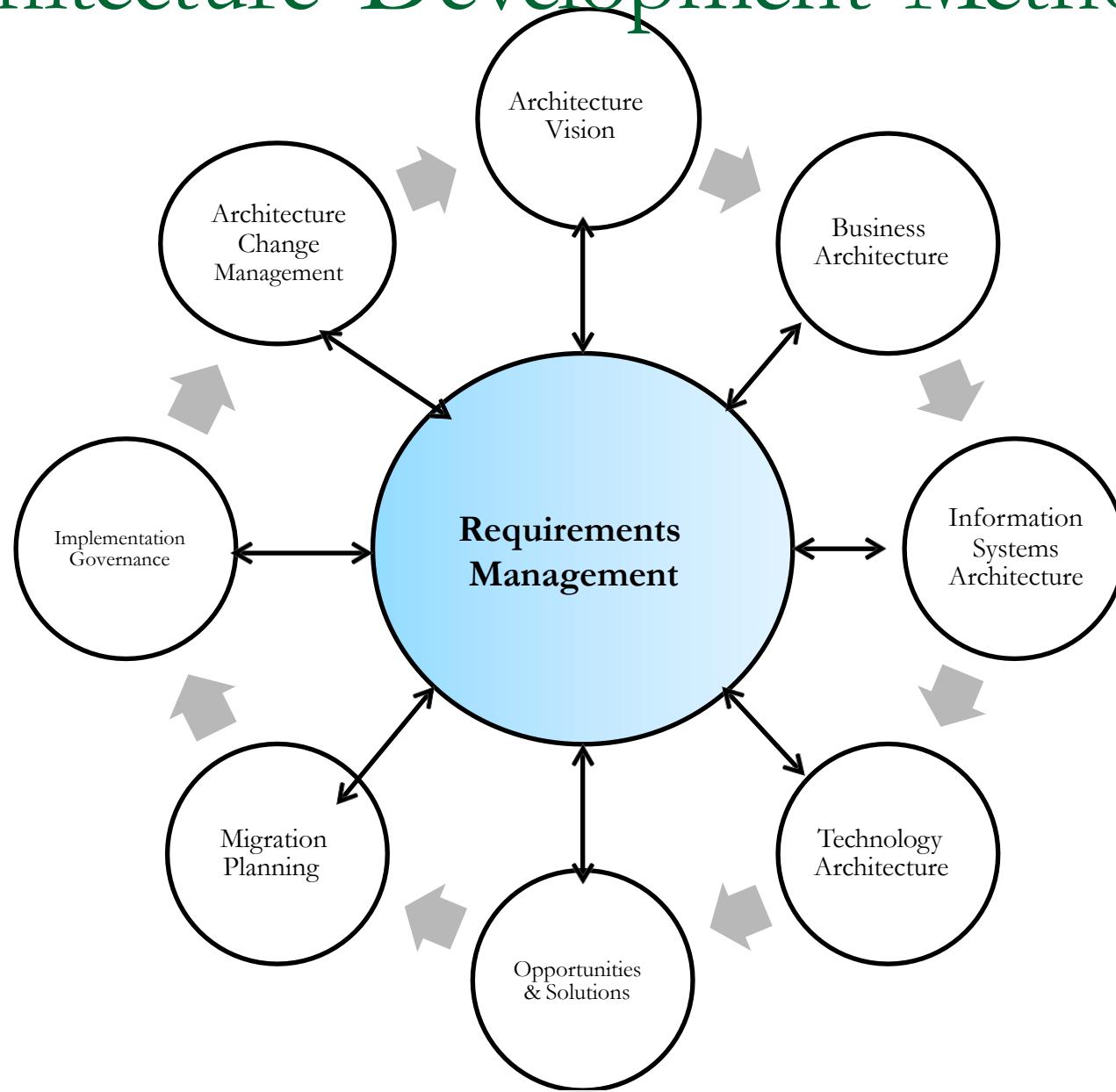
ADL – ACME from CMU



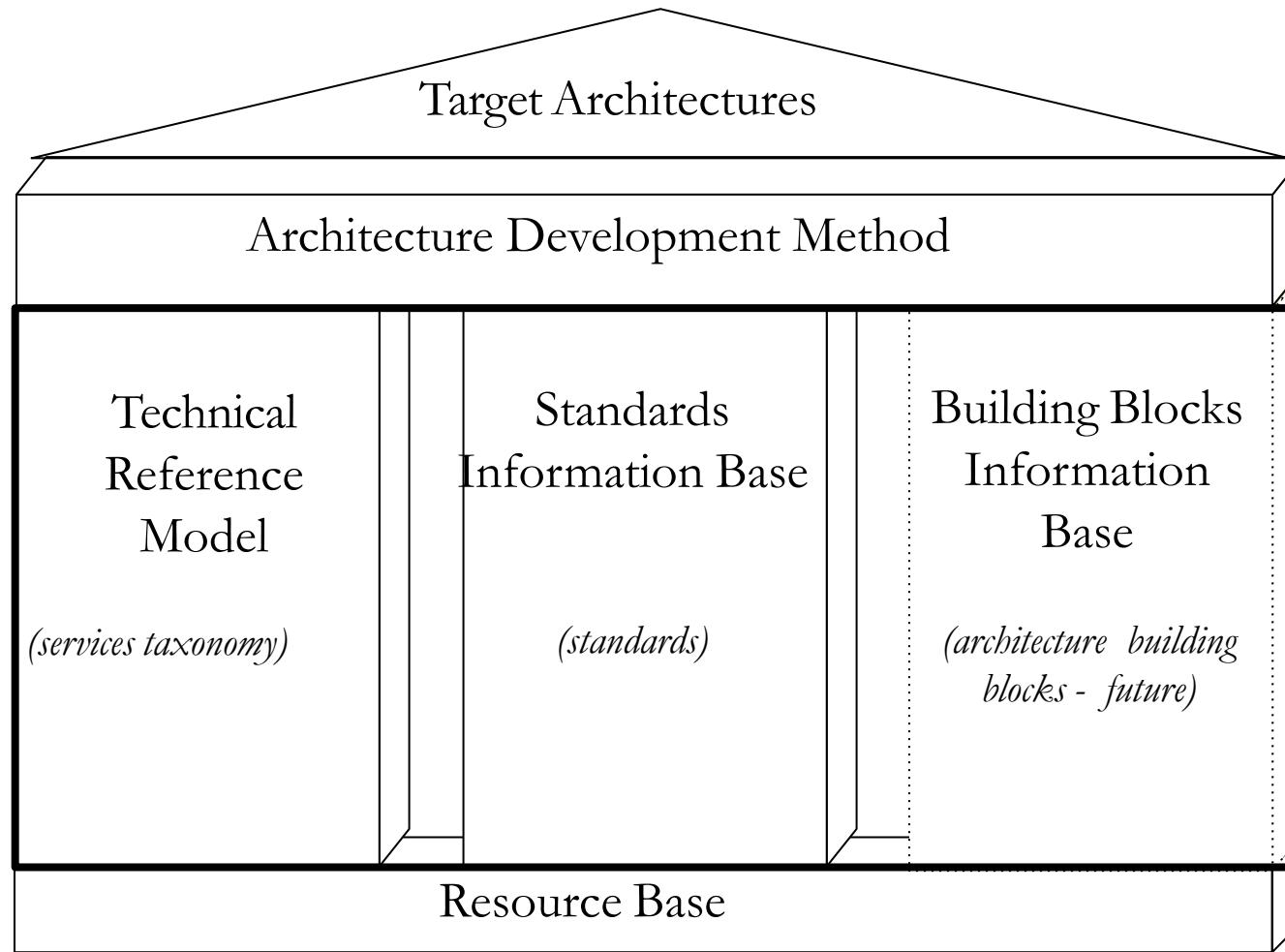
Enterprise Architecture - TOGAF

- Enterprise Architecture
 - Business Architecture
 - Information and Data Architecture
 - Application Architecture
 - Technology Architecture
- Architecture Development Method (ADM)
- Reference Architecture

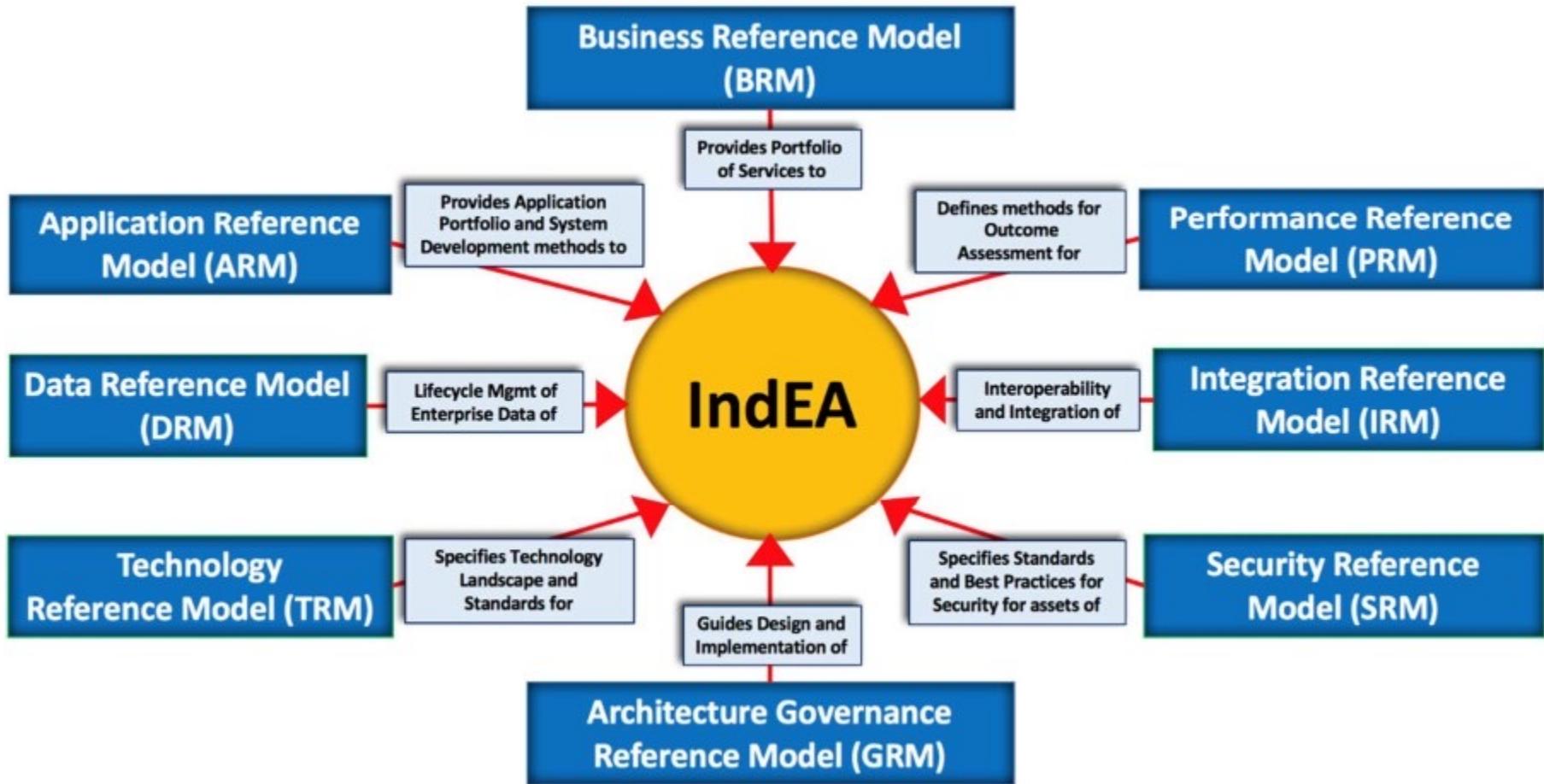
Architecture Development Method



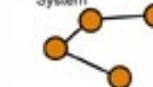
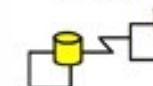
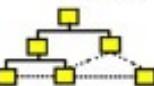
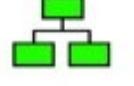
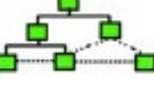
TOGAF Foundation Architecture



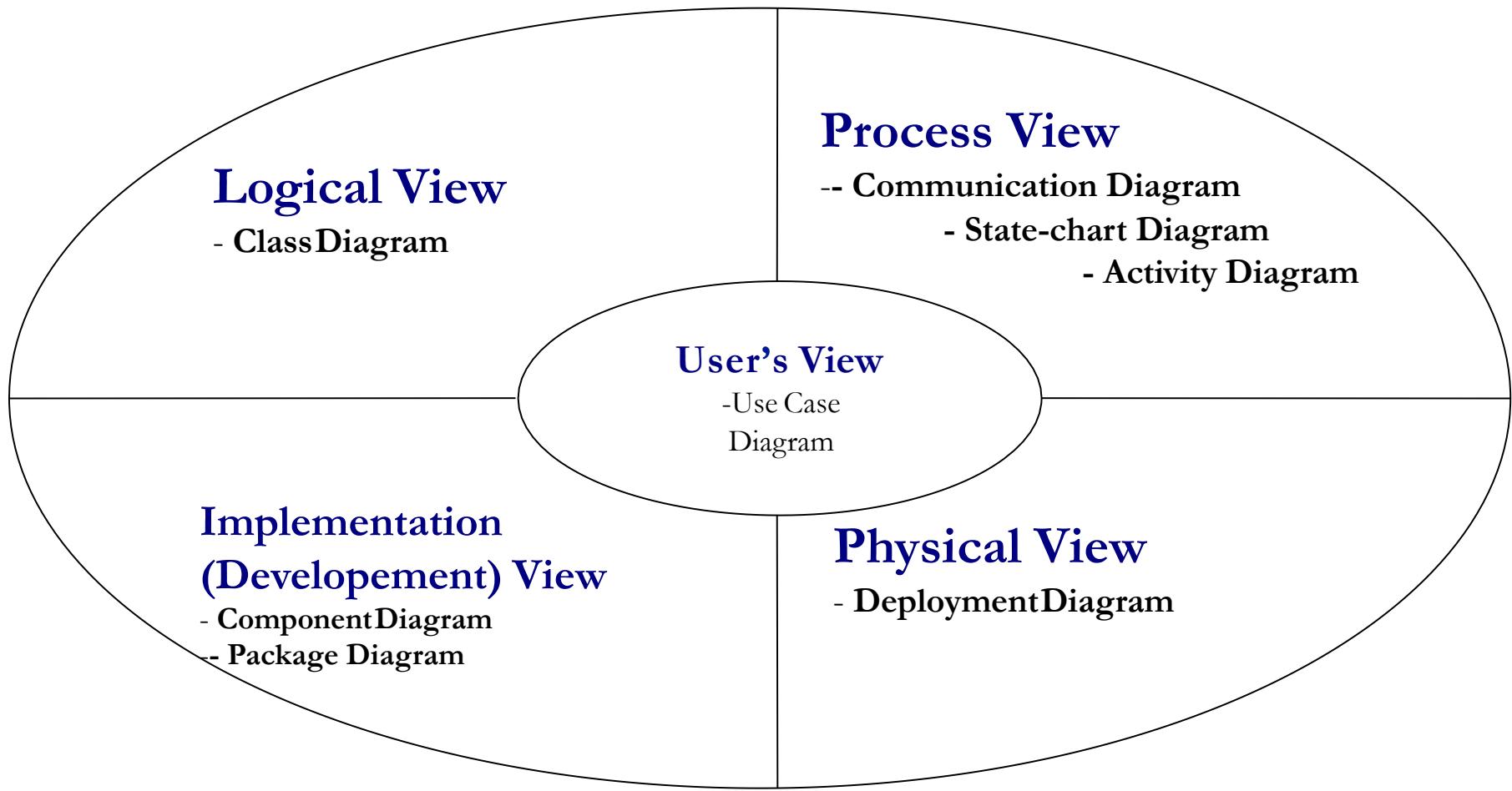
India Reference Architecture Model



Zachman Framework

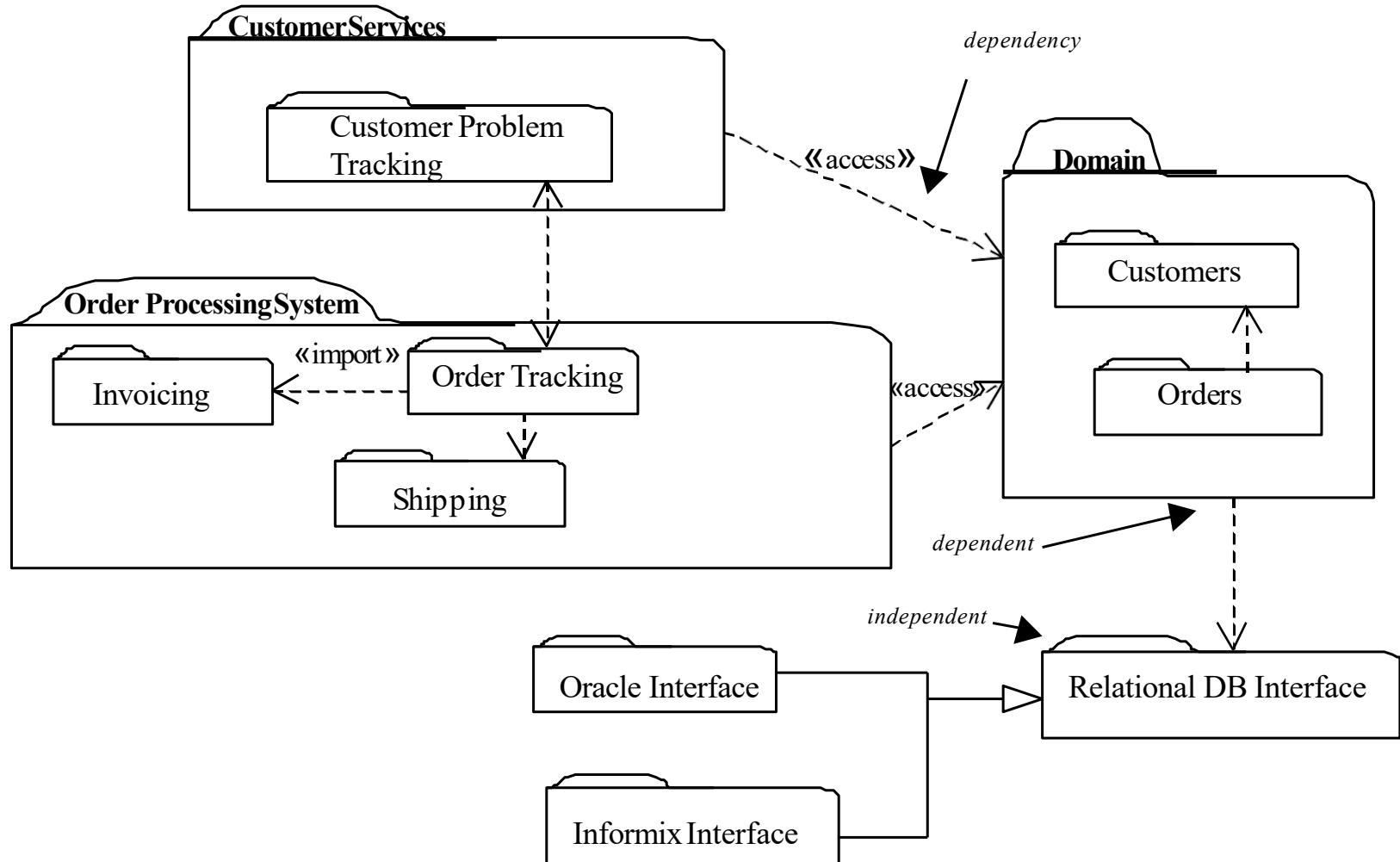
	DATA What	FUNCTION How	NETWORK Where	PEOPLE Who	TIME When	MOTIVATION Why	
SCOPE (CONTEXTUAL)	List of Things Important to the Business 	List of Processes the Business Performs 	List of Locations in which the Business Operates  Node = Major Business Location	List of Organizations Important to the Business  People = Major Organization Unit	List of Events/Cycles Significant to the Business  Time = Major Business Event/Cycle	List of Business Goals/Strategies  Ends/Means = Major Business Goal/Strategy	SCOPE (CONTEXTUAL)
Planner	ENTITY = Class of Business Thing	Process = Class of Business Process					Planner
BUSINESS MODEL (CONCEPTUAL)	e.g. Semantic Model 	e.g. Business Process Model 	e.g. Business Logistics System  Node = Business Location Link = Business Linkage	e.g. Work Flow Model  People = Organization Unit Work = Work Product	e.g. Master Schedule  Time = Business Event Cycle = Business Cycle	e.g. Business Plan  End = Business Objective Means = Business Strategy	BUSINESS MODEL (CONCEPTUAL)
Owner	Ent = Business Entity Reln = Business Relationship	Proc. = Business Process I/O = Business Resources					Owner
SYSTEM MODEL (LOGICAL)	e.g. Logical Data Model 	e.g. Application Architecture 	e.g. Distributed System Architecture  Node = I/S Function (Processor, Storage, etc.) Link = Line Characteristics	e.g. Human Interface Architecture  People = Role Work = Deliverable	e.g. Processing Structure  Time = System Event Cycle = Processing Cycle	e.g., Business Rule Model  End = Structural Assertion Means = Action Assertion	SYSTEM MODEL (LOGICAL)
Designer	Ent = Data Entity Reln = Data Relationship	Proc. = Application Function I/O = User Views					Designer
TECHNOLOGY MODEL (PHYSICAL)	e.g. Physical Data Model 	e.g. System Design 	e.g. Technology Architecture  Node = Hardware/Systems Software Link = Line Specifications	e.g. Presentation Architecture  People = User Work = Screen Format	e.g. Control Structure  Time = Execute Cycle = Component Cycle	e.g. Rule Design  End = Condition Means = Action	TECHNOLOGY MODEL (PHYSICAL)
Builder	Ent = Segment/Table/etc. Reln = Pointer/Key/etc.	Proc. = Computer Function I/O = Data Elements/Sets					Builder
DETAILED REPRESENTATIONS (OUT-OF-CONTEXT)	e.g. Data Definition 	e.g. Program 	e.g. Network Architecture  Node = Address Link = Protocol	e.g. Security Architecture  People = Identity Work = Job	e.g. Timing Definition  Time = Interrupt Cycle = Machine Cycle	e.g. Rule Specification  End = Sub-condition Means = Stop	DETAILED REPRESENTATIONS (OUT-OF-CONTEXT)
Sub-Contractor	Ent = Field Reln = Address	Proc. = Language Statement I/O = Control Block					Sub-Contractor
FUNCTIONING ENTERPRISE	e.g. DATA	e.g. FUNCTION	e.g. NETWORK	e.g. ORGANIZATION	e.g. SCHEDULE	e.g. STRATEGY	FUNCTIONING ENTERPRISE

4 + 1 Architectural view



Views mapped to diagram types available in UML

Development (Implementation) view

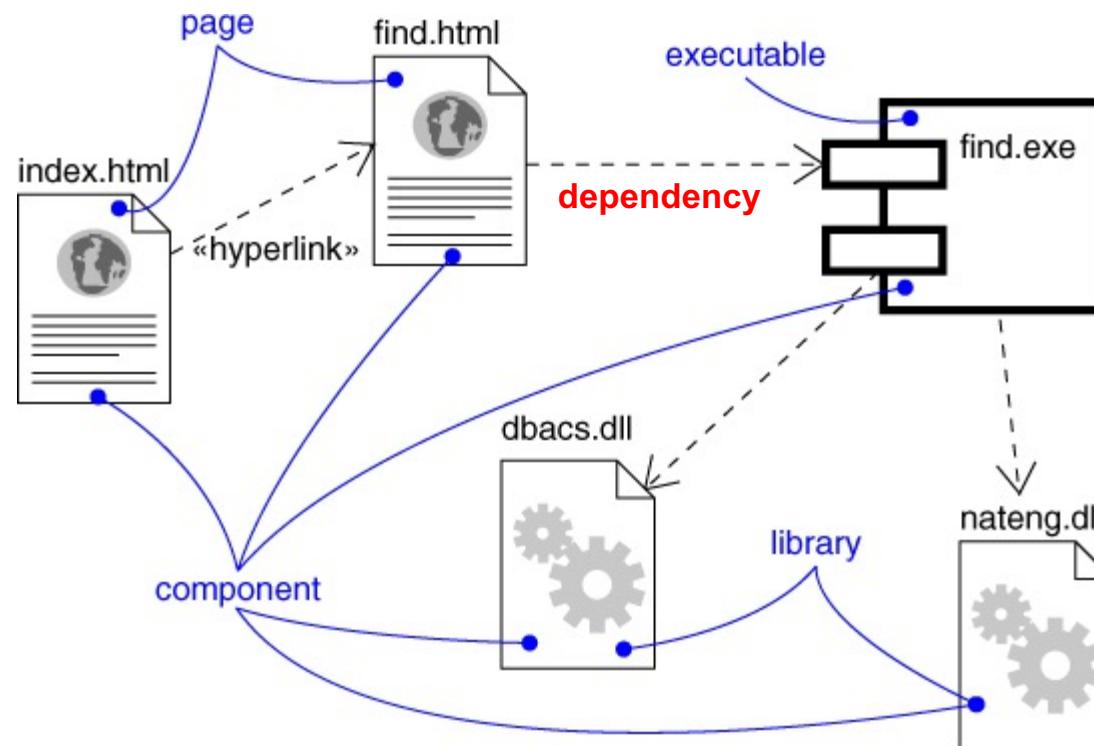


Component Diagram

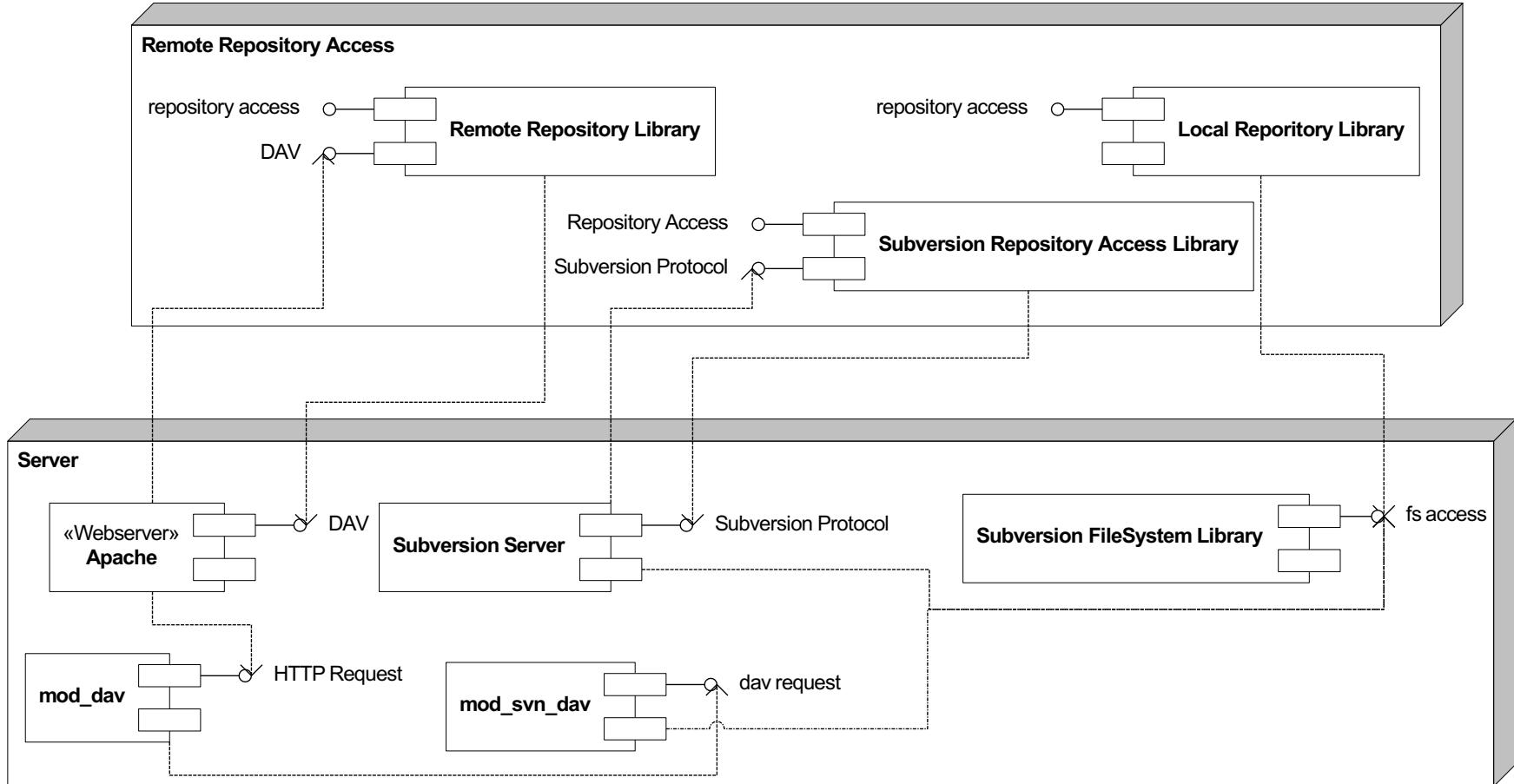
- Captures the physical structure of the implementation (code components)

Components:

- Executables
- Library
- Table
- File
- Document

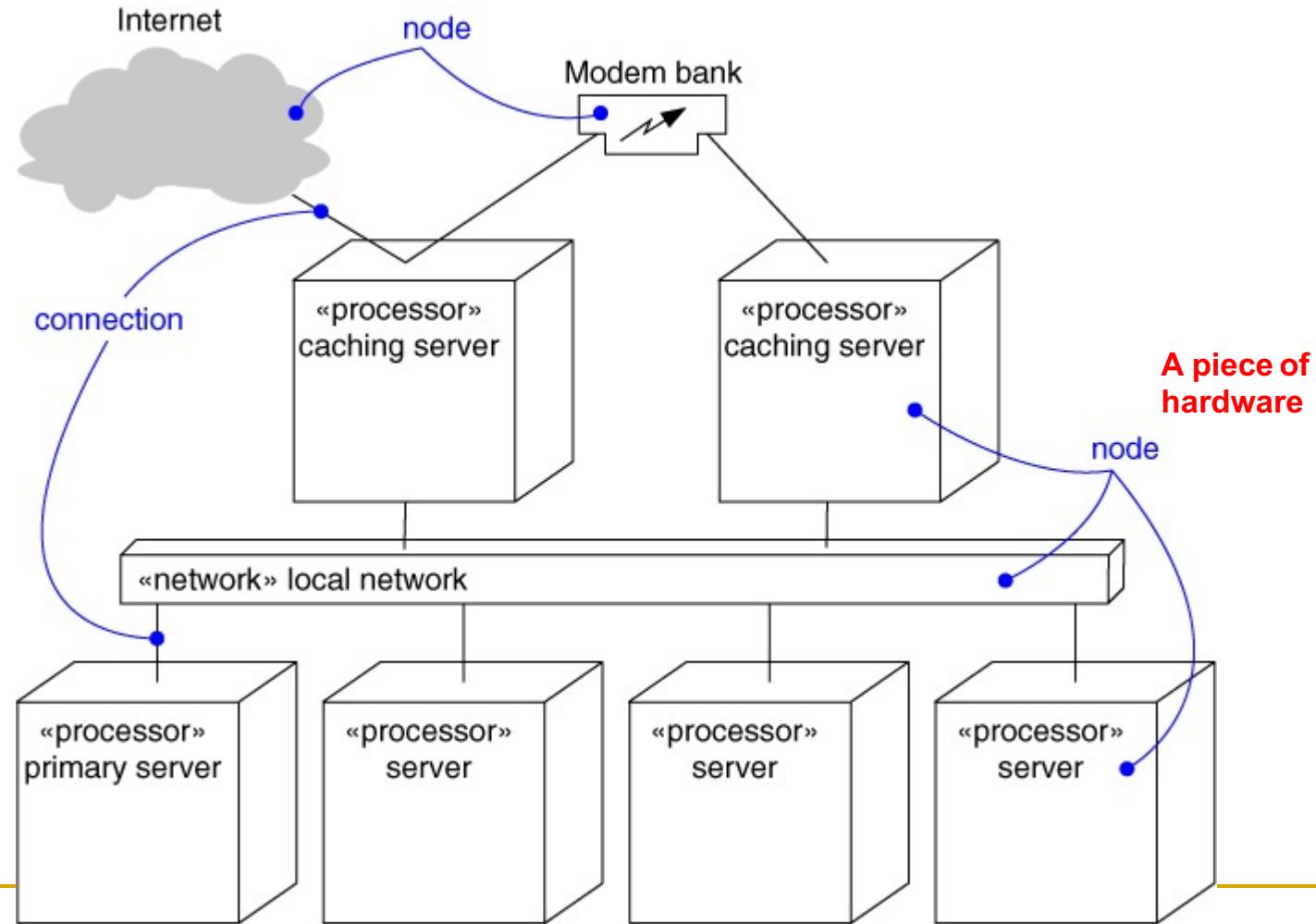


Component and Connector – Implementation View

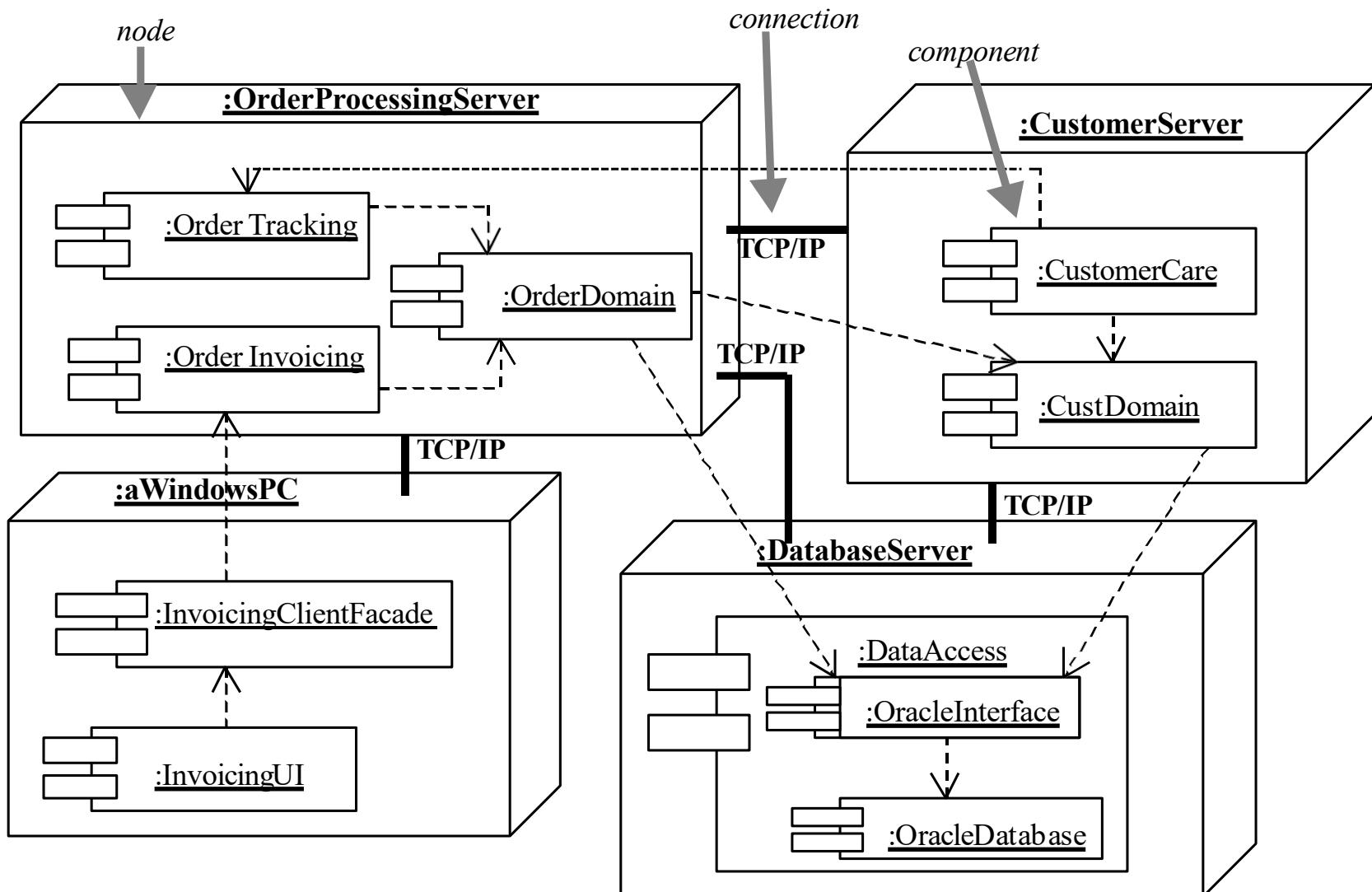


Deployment view

- Captures the topology of a system's hardware

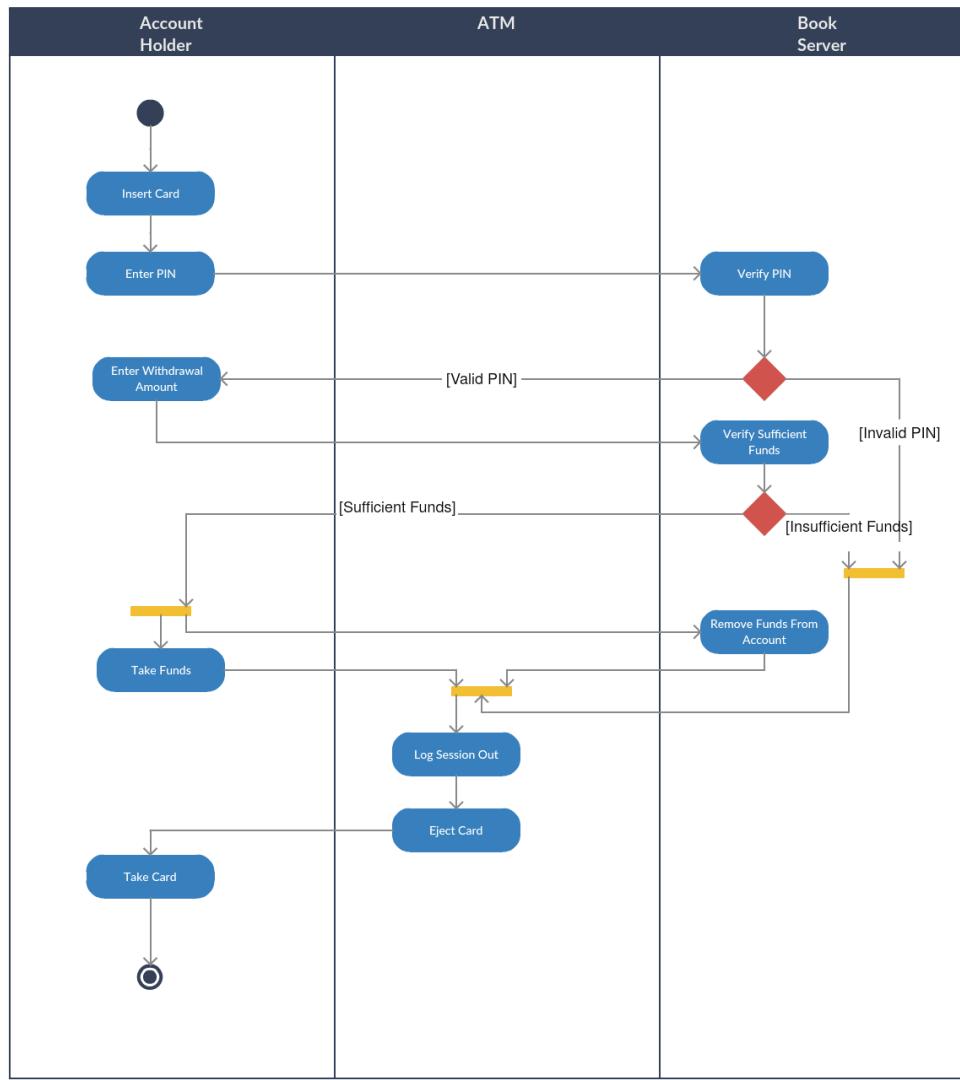


Deployment (Physical) View



Process view

ATM SYSTEM for ABC BANK



Can be described using
State diagrams, Activity
diagrams, Communication
diagrams

Designing the Architecture

Do you see Antipatterns?
It is time to refactor /rearchitect !



Do you see Antipatterns?
It is time to refactor /rearchitect !



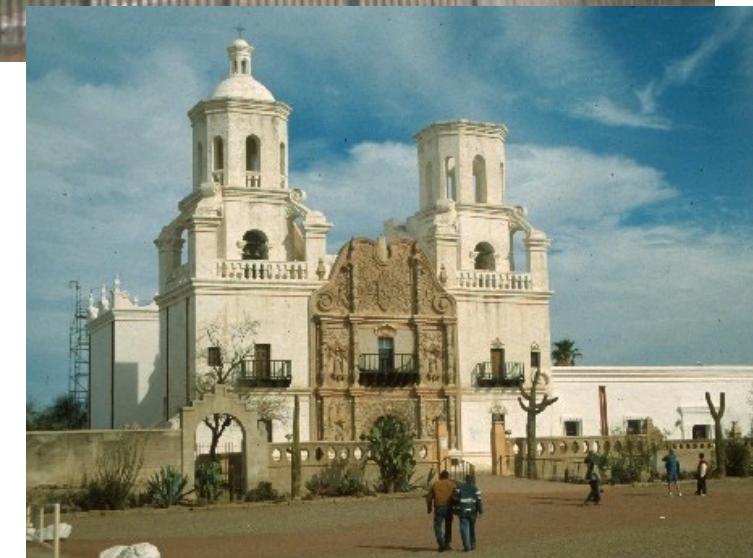
Ray and Maria Stata Center for Computer, Information, and Intelligence Sciences, MIT

Anti-patterns

Antipatters → Inappropriate solution

- Design by committee
- Abstract Inversion
- Vendor Lock-in
- Warm bodies
- Cover your assets
- Reinvent the Wheel
- Interface bloat
- Ambiguous viewpoint
- Big ball of mud
- Gold plating
- Inner-platform effect
- Magic Pushbutton
- Race hazard
- Stovepipe

What is an Architectural Style?

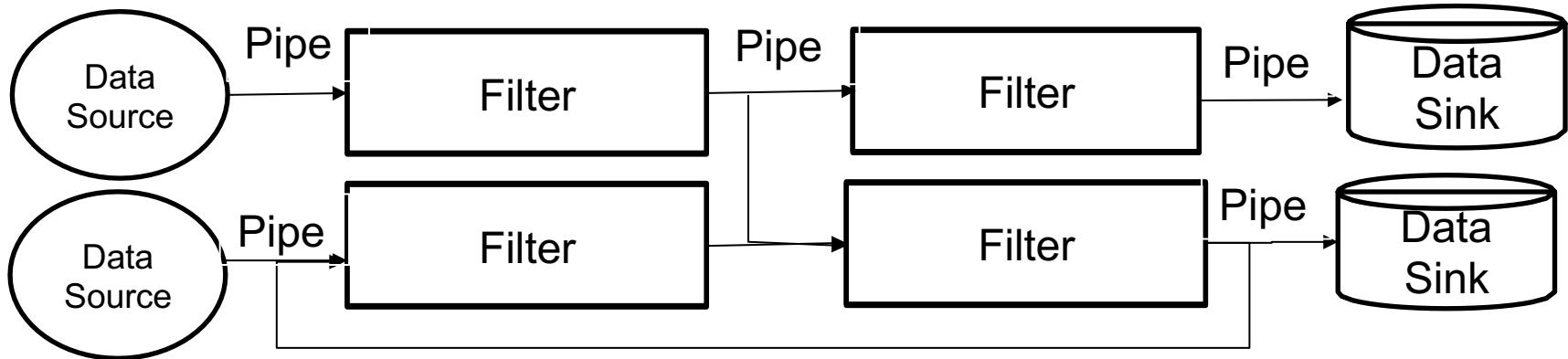


Architectural Patterns

- Pipe & Filter
- Blackboard
- Distributed
- Layered
 - Microkernel
 - Client Server
 - MVC
 - Microservices
- Batch
- Repository-Centric
- IR-Centric
- Subsumption
- Bridge
- Reflection
- Event Driven

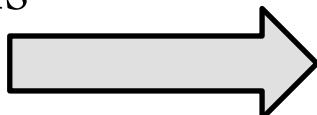
Architecture Pattern - Pipe and Filter

- Inspired by Unix architecture, Data driven interaction
- Data source/pump is connected to data filters by pipes
- Final data is received by data sink/consumer

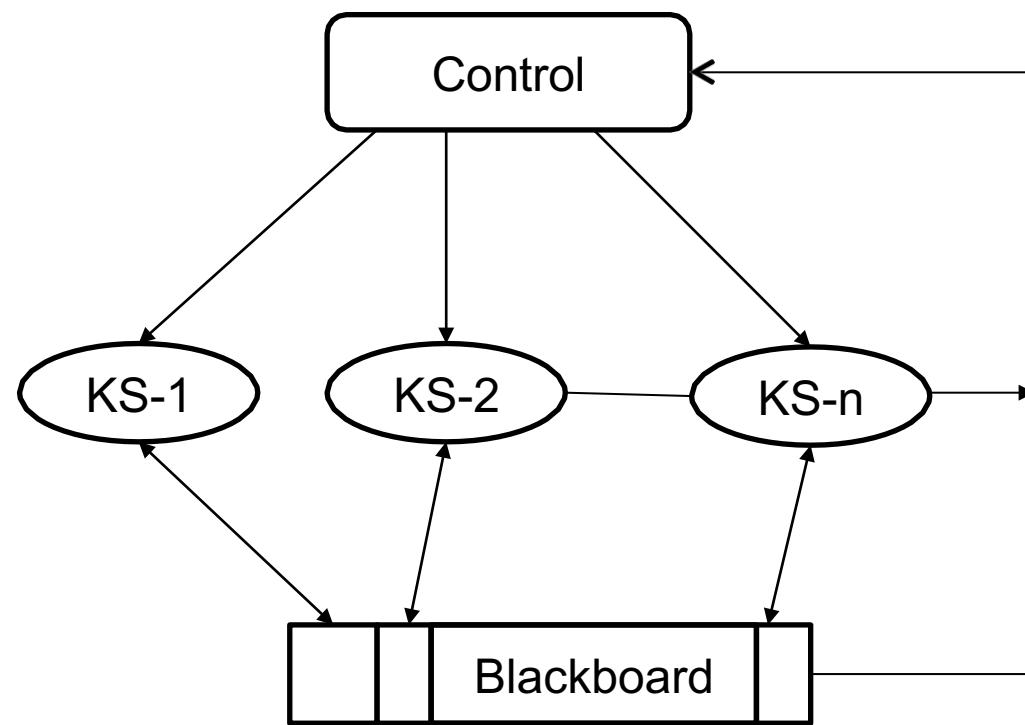


- Difficulty of error handling
- Sharing state information is expensive / inflexible
- Efficiency Loss
 - Cost of transferring and transforming data
 - Data dependencies
 - Cost of context switching

Limitations



Architecture Pattern - Blackboard

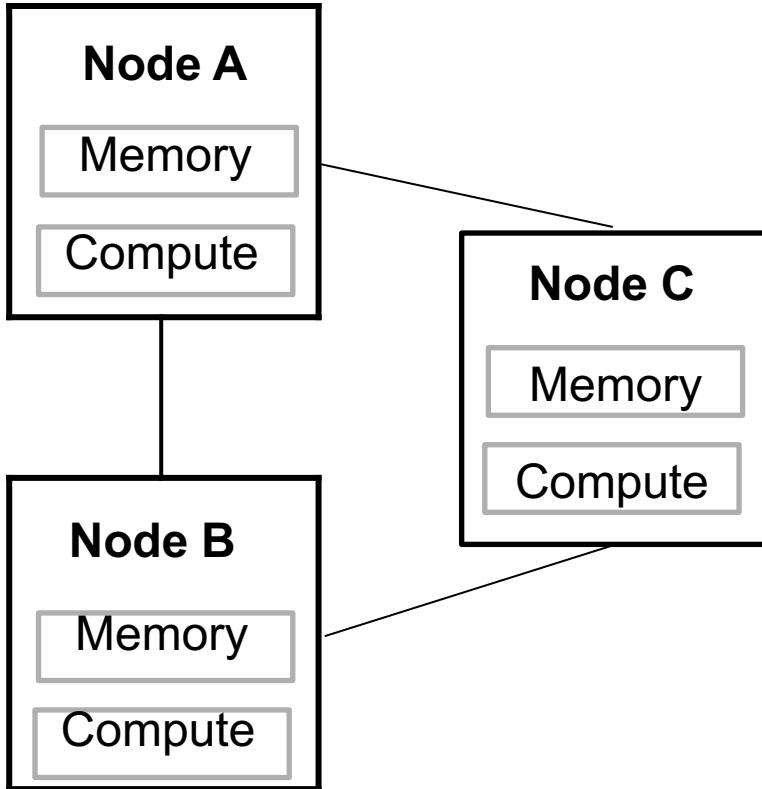


Limitations

- Problems in synchronization
- Difficult to decide when to terminate
- Significant impact on all of its agents

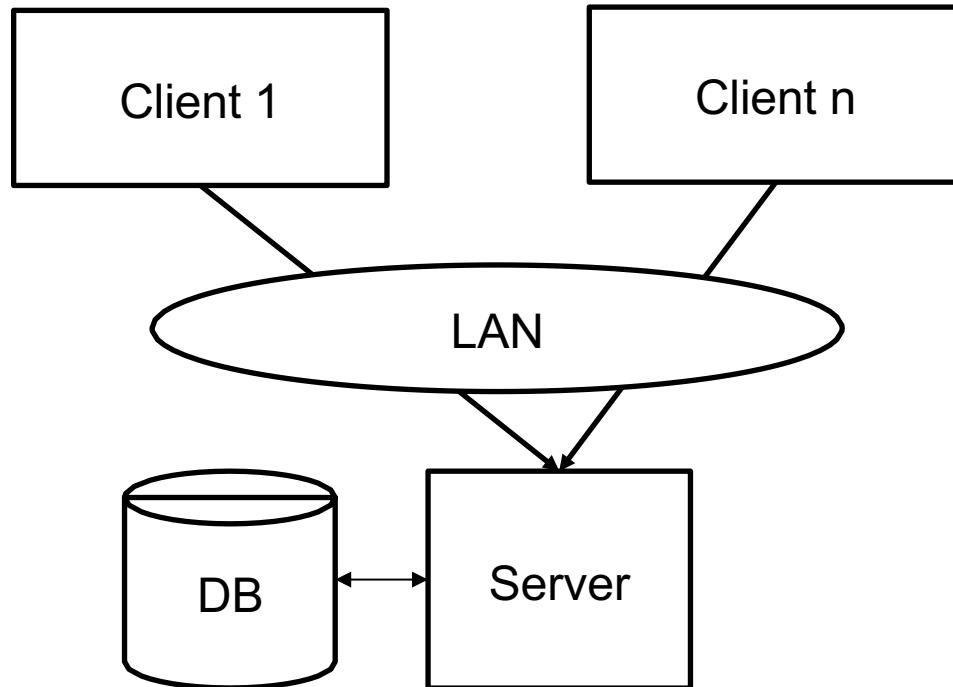
- Integrate large and diverse specialized modules, implement complex, non deterministic control strategies.
- Blackboard is a structured global memory containing objects from solution space
- Knowledge sources can be seen as highly specialized modules with their own representation.
- Control component selects, configures and executes knowledge sources.

Distributed Systems



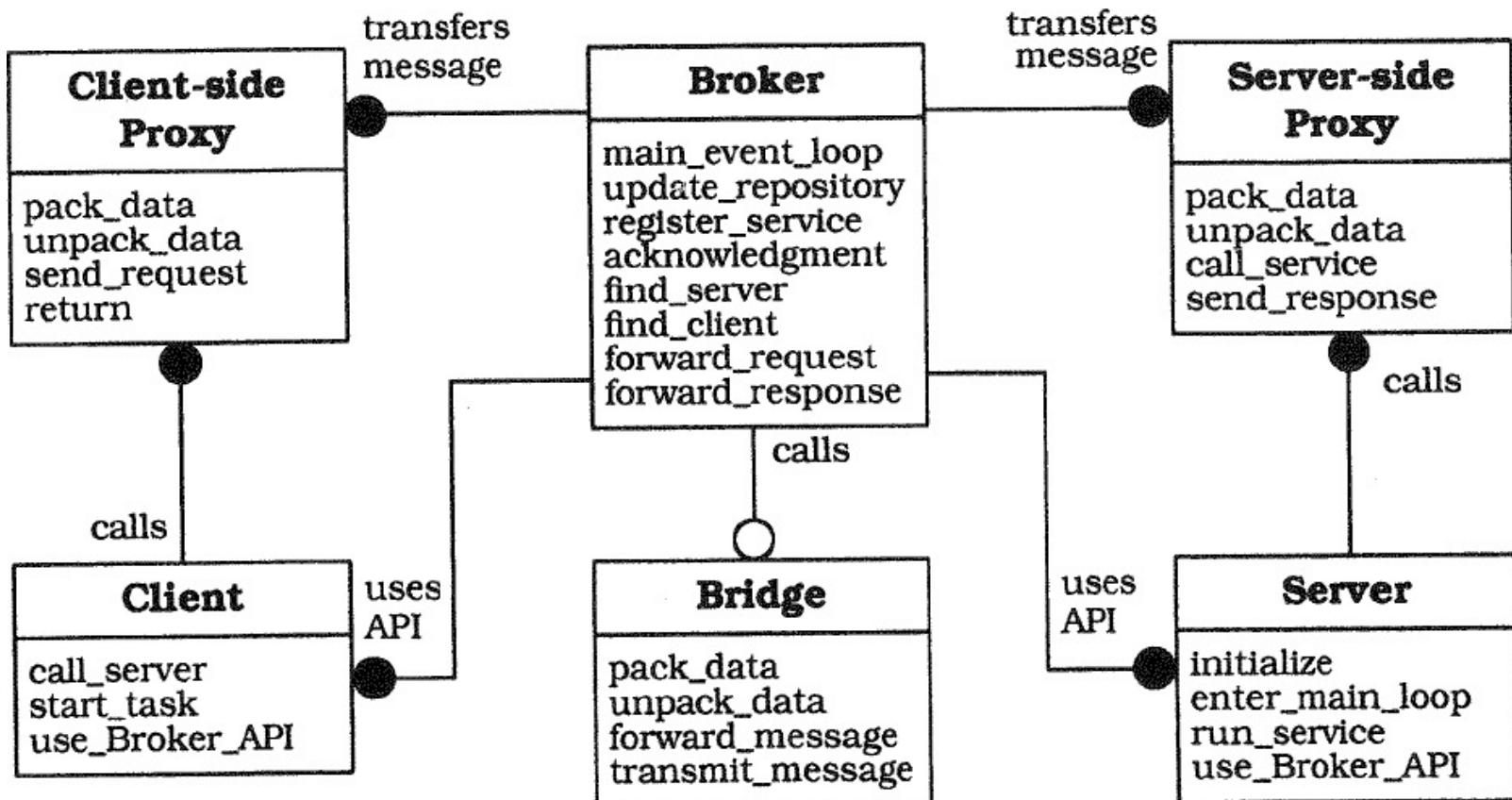
- Structure distributed software systems with decoupled components that interact by remote service invocation
- Requires concurrent components, communication network and a synchronization mechanism
- Allows resource sharing, including software by systems connected to the network
- Client-Server, Broker, Star & Ring
- A broker component is responsible for coordinating communication
 - CORBA

Client & Server



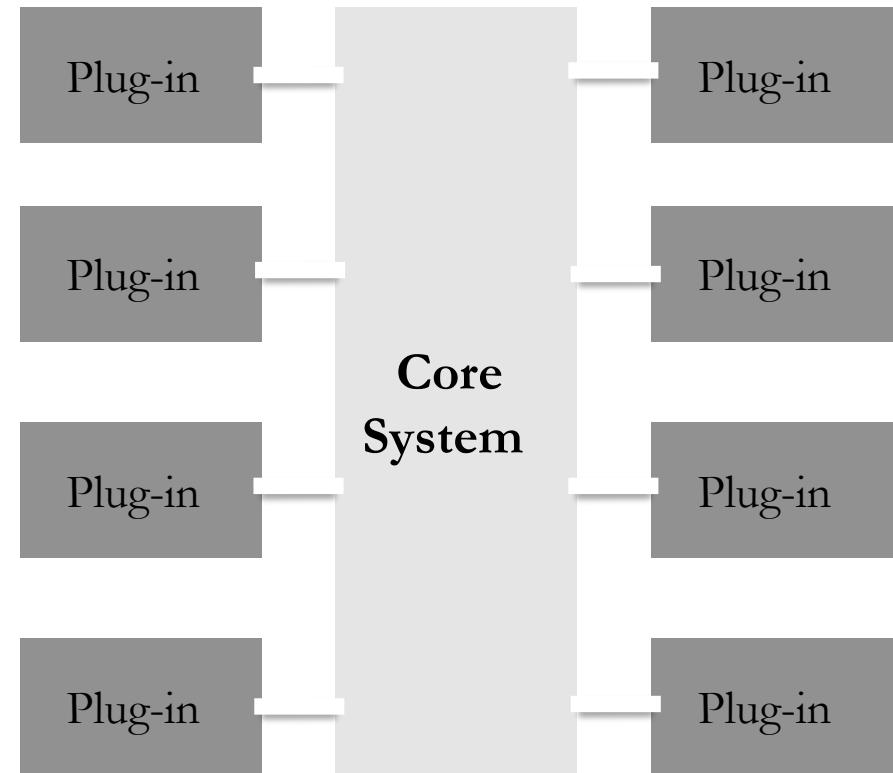
- Message based and modular communications
- A client requests specific services
- Server provides requested services
- Client-Server can reside in the same computer

Broker Pattern



Architecture Pattern - Microkernel

- Software systems that must be able to adapt to changing system requirements.
- Separates a minimal functional core from extended functionality and customer-specific parts
- Contracts between plug-in modules and core system can range anywhere from standard contracts to custom ones

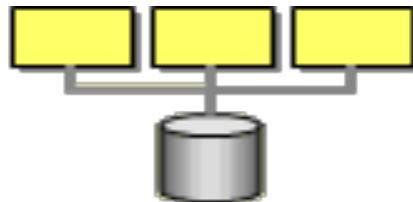


Limitations → Scalability and Ease of development

Architectural Patterns for Integration..



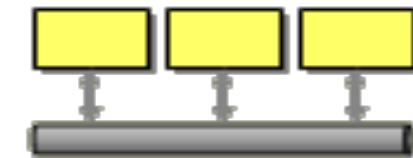
File Transfer



Shared Database



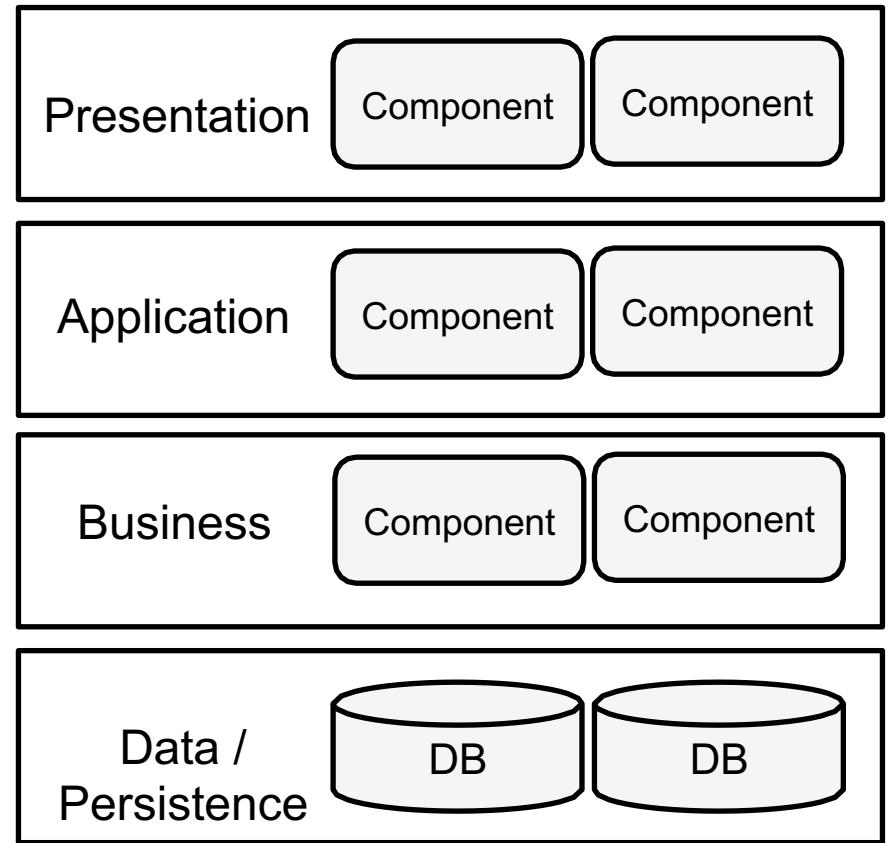
Remote Procedure



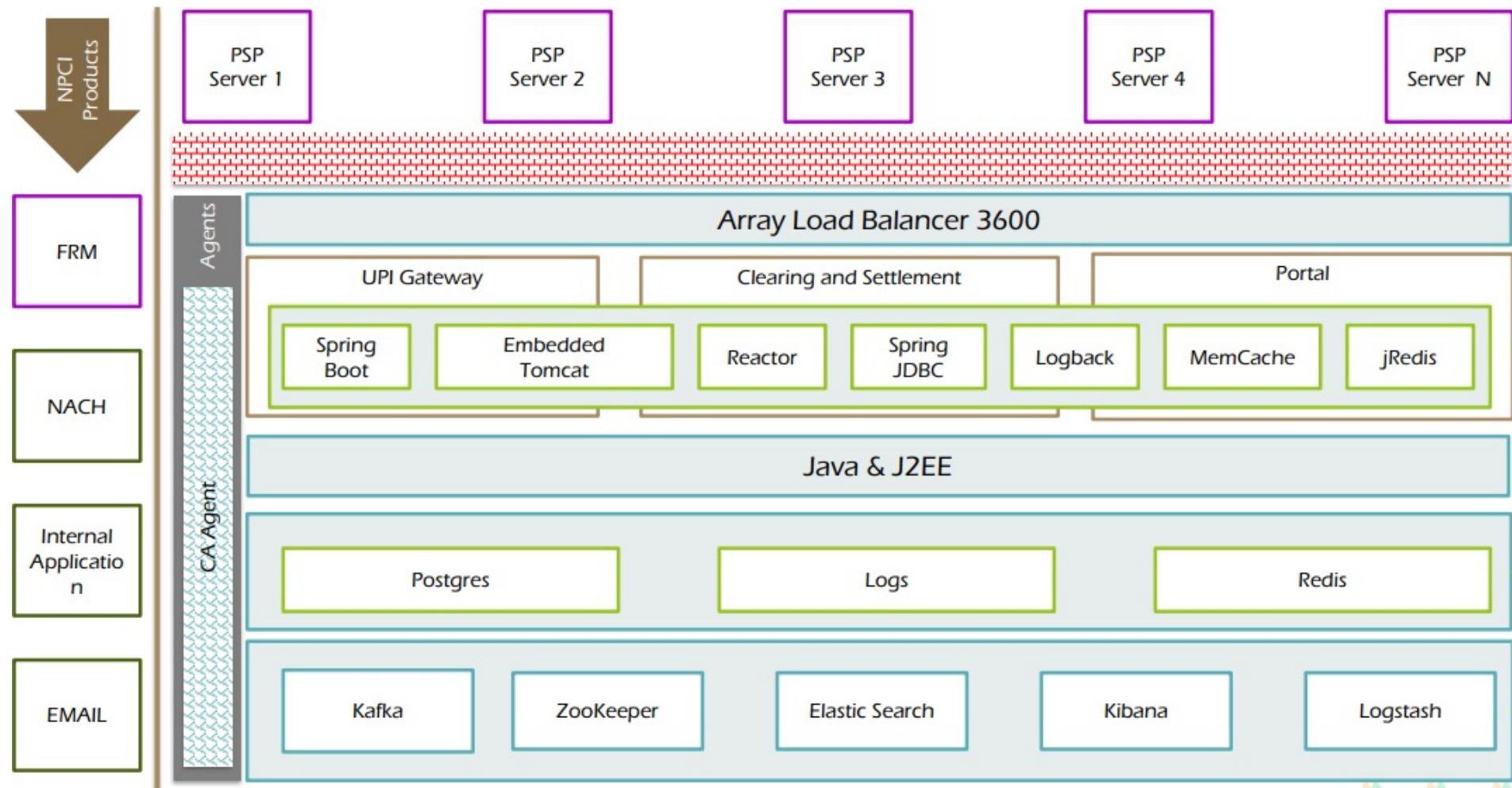
Messaging

Layered Pattern

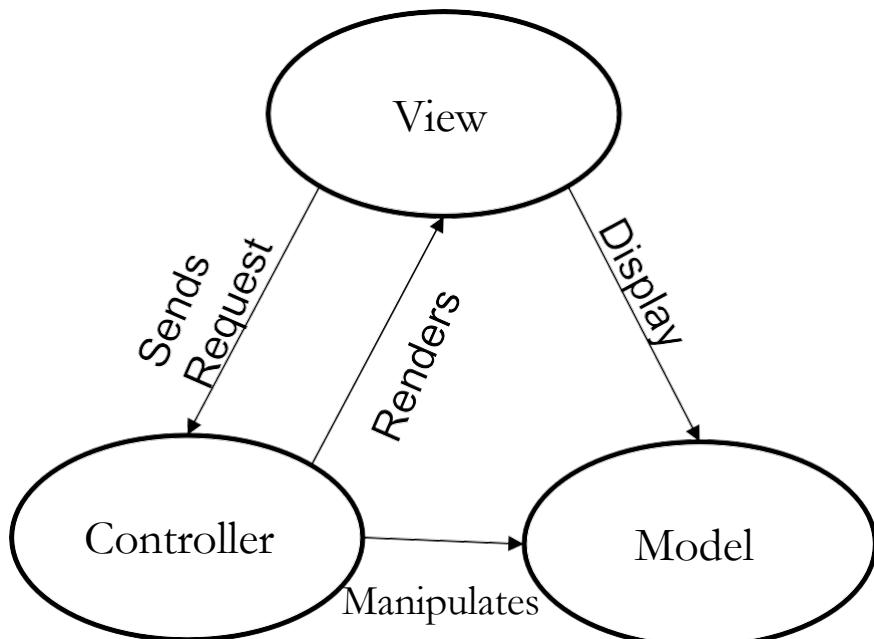
- Most common modern architecture, known as n-tier architecture
- Separation of concerns among components
- Layers of isolation is another key concept



Mission Critical System - UPI



Architecture Pattern - MVC

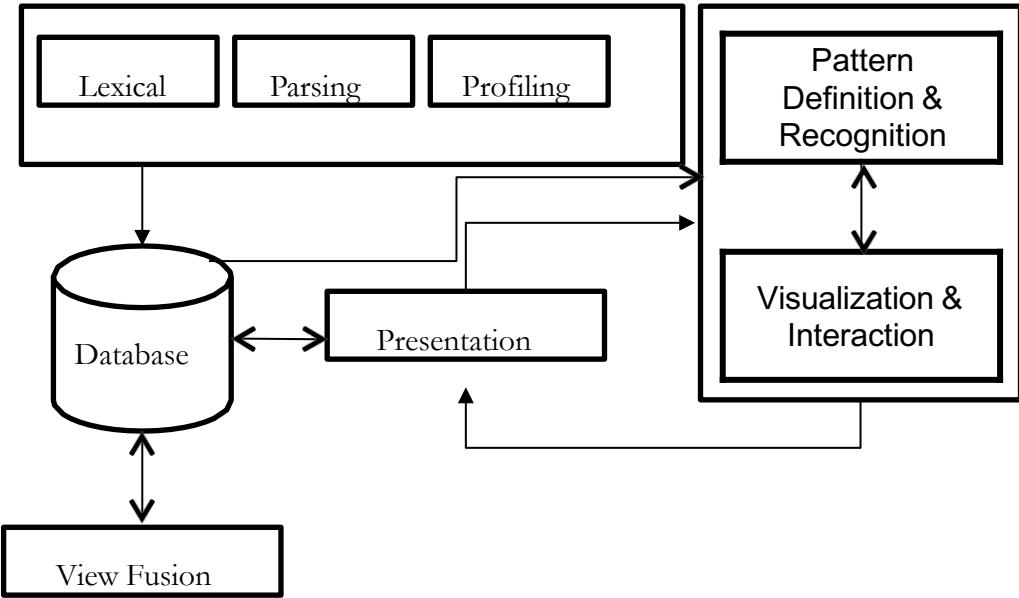


- **Model** is responsible for data and logic: such as persistence, model objects, parsers and networking
- **View** is responsible for objects that are in charge of the visual representation (UI) of Model and controls the user interacts with.
- **Controller** acts as an intermediary between application's view objects and its model objects
- Variations - MVP, MVVM, PMVC

Reconstructing Architecture

■ Information extraction:

- Collect data
- Analyze artifacts – code, documents & execution traces
- Extraction - Files, Functions & Variables
- Program Comprehension
- Validate results



■ Database construction:

- Convert to standard form
- Design database to optimize queries
- Consider saving most used queries as separate tables

■ View fusion:

- Combine information – fill in missing information, resolve ambiguities & correct errors

■ Reconstruction: Apply abstraction

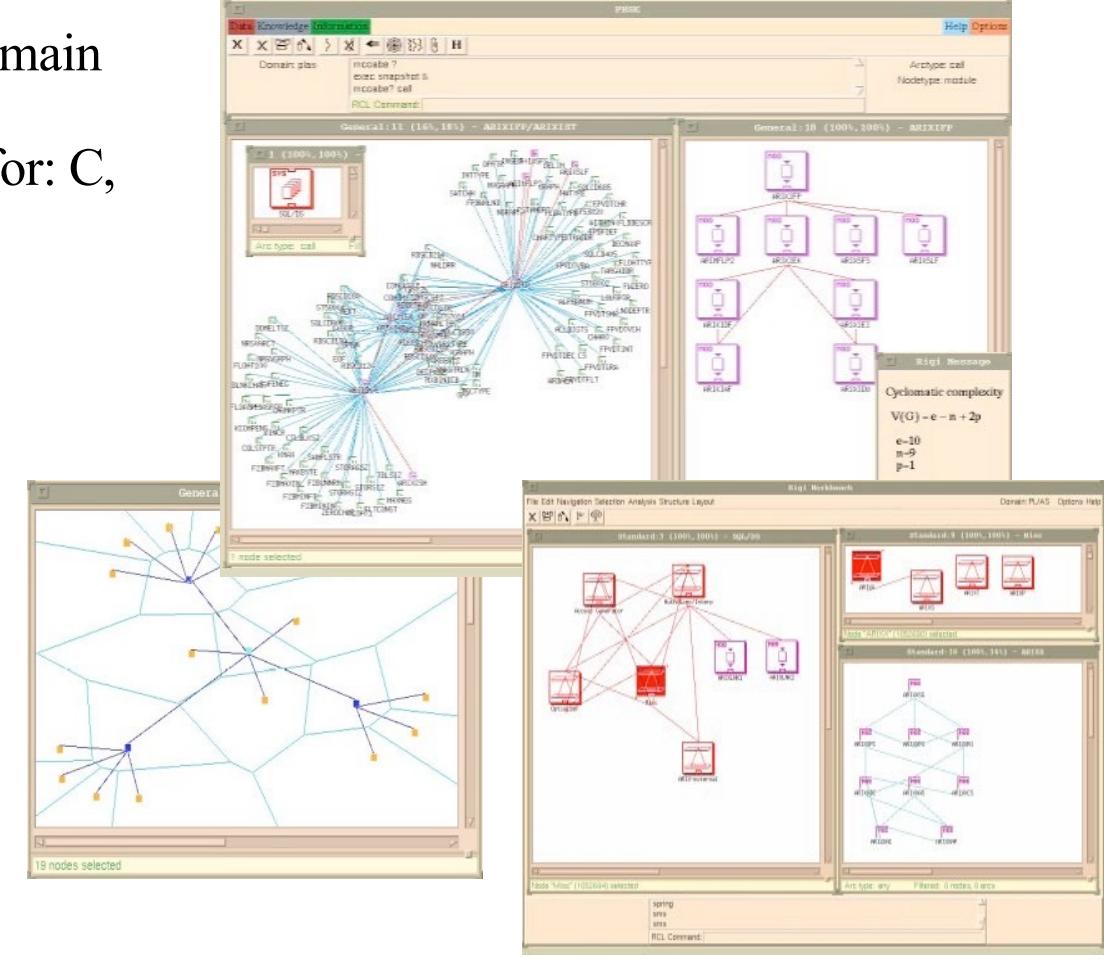
Reconstruction - Visualization

- Scope
- Abstraction
- Specification Methods
- Presentation
- Tools
 - Static Visualization
 - Code Crawler, Rigi, Seesoft, Sniff++, Revolve
 - Dynamic Visualization
 - Imagix4D, Polka

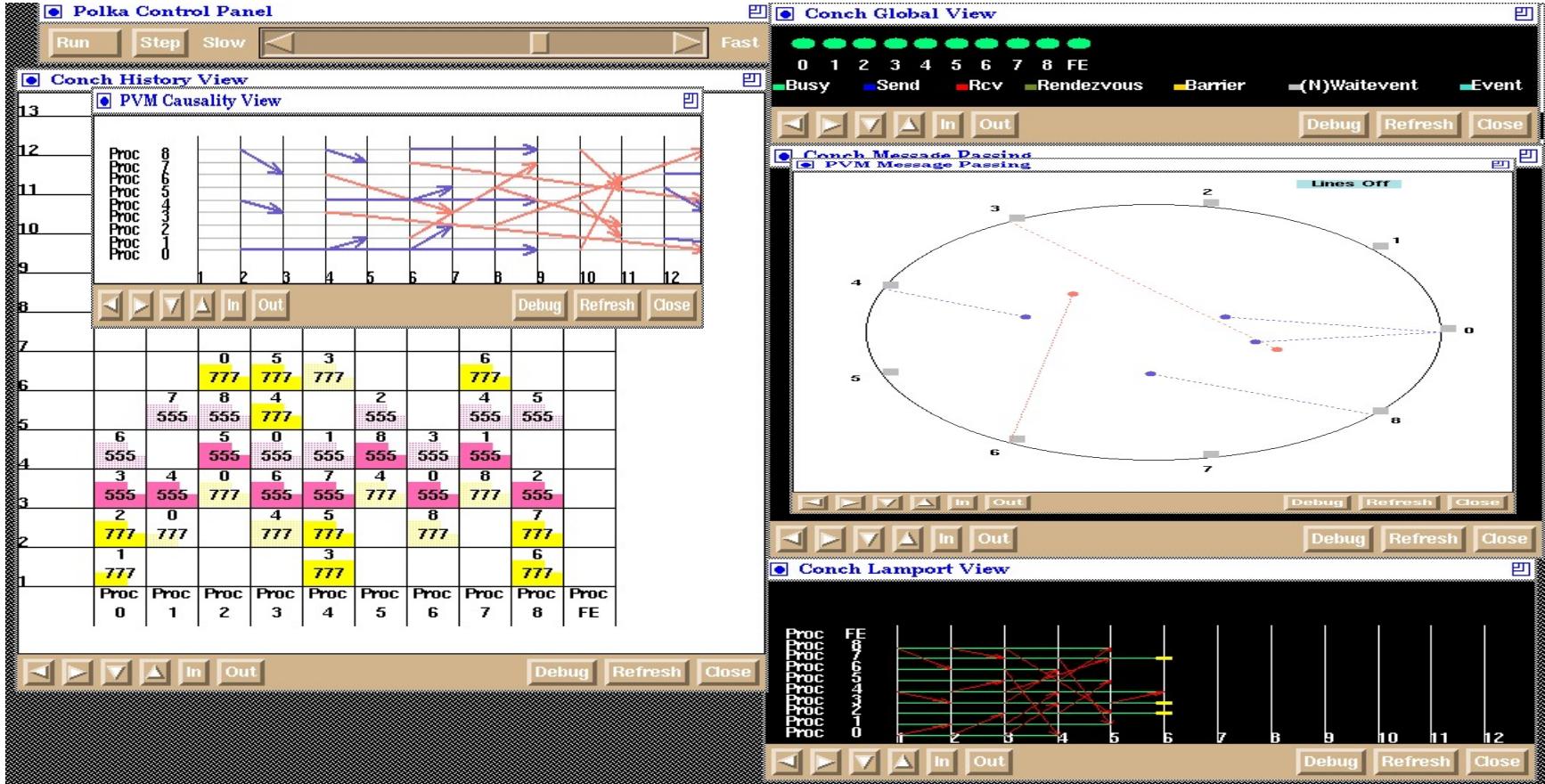
Software Visualization – Static - Rigi

- A research and public domain visualization tool, user programmable, analysis for: C, C++, COBOL, PL/AS, LaTeX

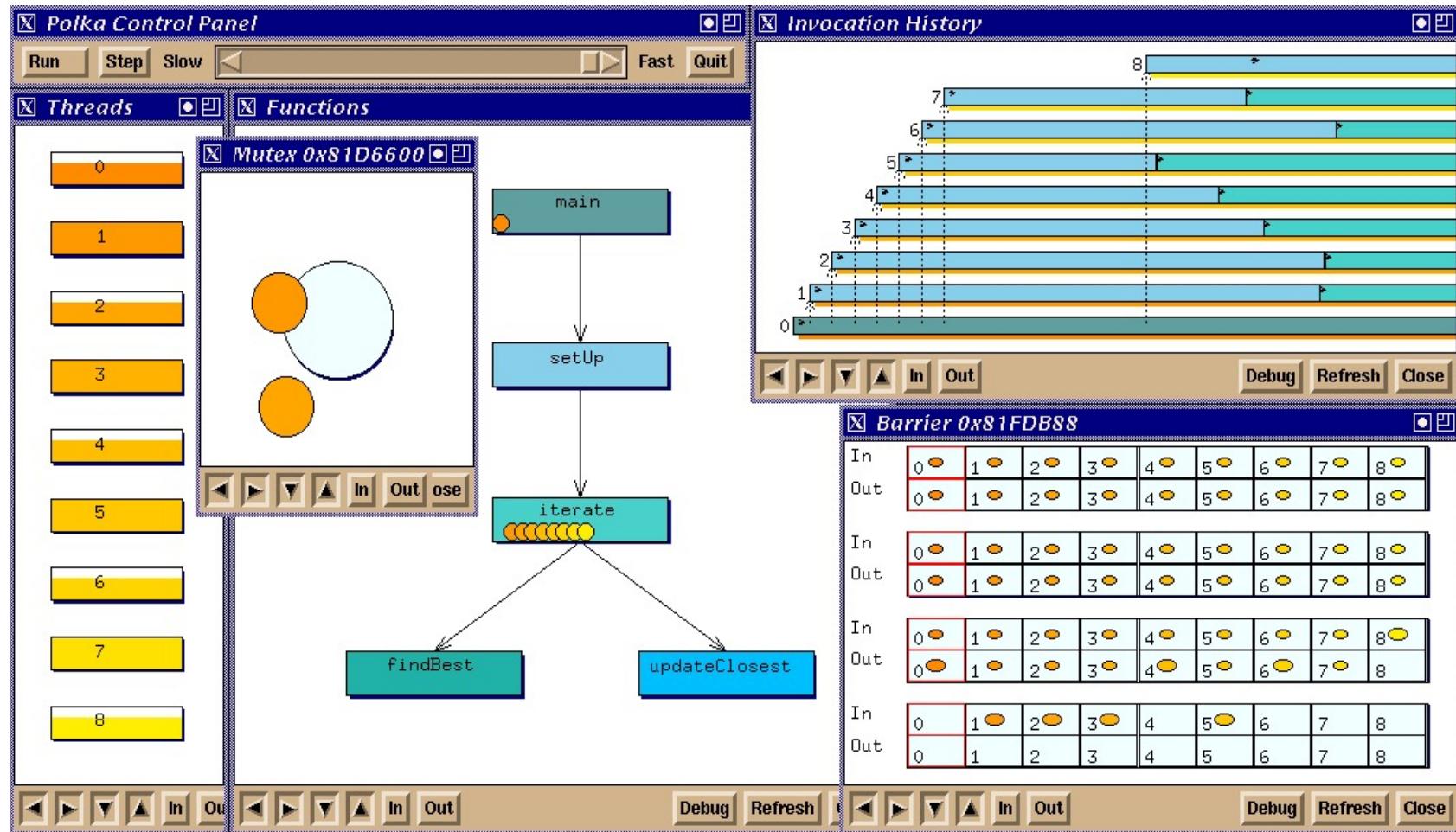
Artifact	Icon representation
system	
subsystem	
module	
proc	
data	
struct	
member	



Software Visualization – Dynamic - POLKA

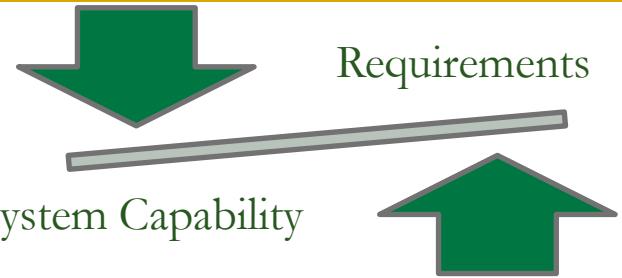


Message Passing View, History View,



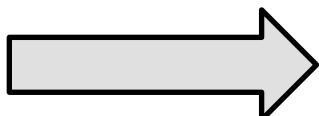
Threads View, Function View, History View, Mutex View, Barrier View

Architecture Evaluation



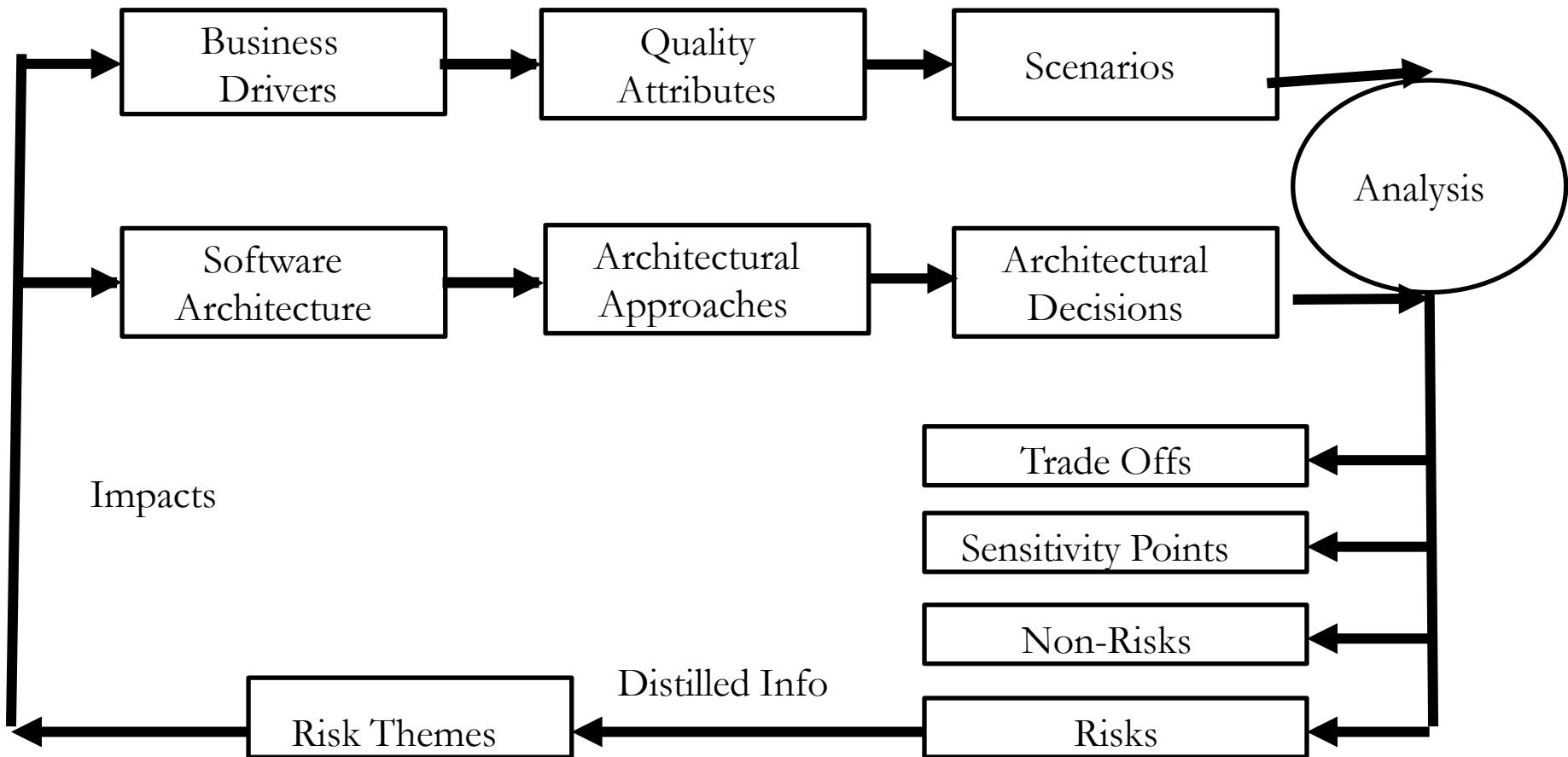
Review Method	Generality	Level of Detail	Phase	Evaluate
Questionnaire	General	Coarse	Early	Process artifacts
Checklist	Domain	Varies	Middle	Process artifacts
Scenarios	System	Medium	Middle	Artifacts
Metrics	General / Domain	Fine	Middle	Artifacts
Prototype simulation, Experiment	Domain	Varies	Early	Artifacts

Evaluation Techniques



- Software Architecture Analysis Method
- Architecture Trade off and Analysis Method
- Architecture Review for Intermediate Design
- Architecture Level Modifiability Analysis

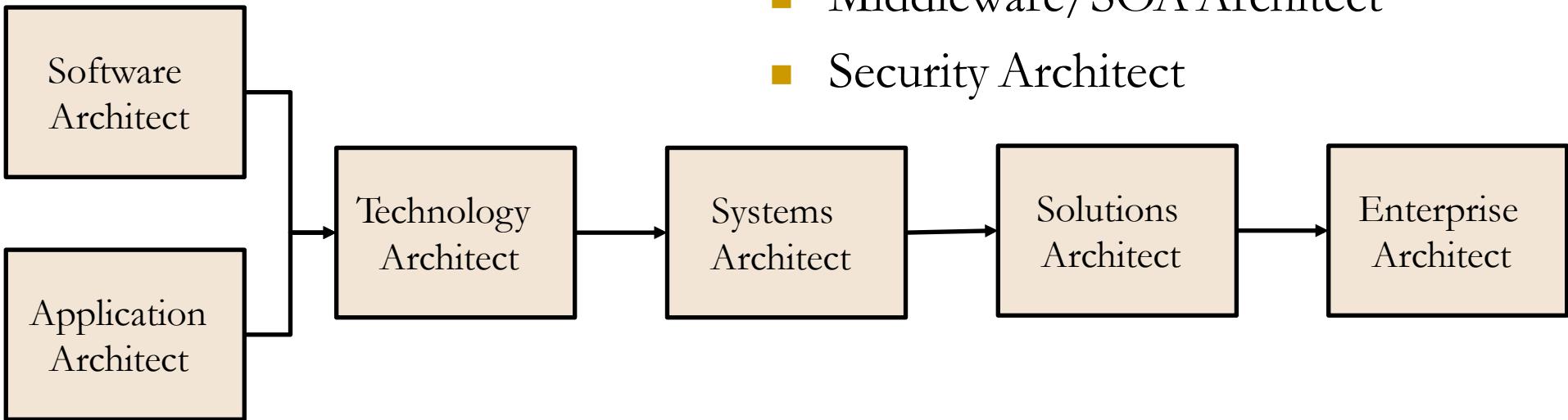
Architecture Evaluation - ATAM



Architecture Related Terms

- Technology Architect
- Software Architect
- Application Architect
- Systems Architect
- Solutions Architect
- Enterprise Architect
- Chief Architect

Various Architects you have dealt with !!!



- Cloud Architect
- Middleware/SOA Architect
- Security Architect

Falls in line with too many definitions on Architecture !!

Quality and Tactics

Quality Scenarios - Table

- Source: who?
 - Some entity (human, computer system, etc.)
- Stimulus: what?
 - Condition that needs to be considered when it arrives at the system
- Environment: when?
 - Stimulus occurs within certain conditions
- Artifact: where?
 - Artifact that is stimulated
- Response: which?
 - Activity undertaken after the arrival of the stimulus
- Measure: how?
 - Measurable in some fashion

Availability

- Failures and faults
- Mean time to failure, repair
- Downtime

Availability Table

- Source: internal, external
- Stimulus: type of fault
- Artifact: processors, channels, storage
- Environment: normal, degraded
- Response: logging, notification, switching to backup, restart, shutdown
- Measure: availability, repair time, required uptime

Availability Scenario example

Availability of the crossing gate controller:

Scenario: Main processor fails to receive an acknowledgement from gate processor.

- Source: external to system
- Stimulus: timing
- Artifact: communication channel
- Environment: normal operation
- Response: log failure and notify operator via alarm
- Measure: no downtime

Availability Tactics

- Fault detection
 - Ping/echo
 - Heartbeat
 - Exception
- Fault recovery
 - Preparation
 - Voting
 - Active redundancy (hot backup)
 - Passive redundancy (warm backup)
 - Spare
 - Re-Introduction
 - Shadow operation
 - State resynchronization
 - Checkpoint/rollback
- Fault prevention
 - Removal from service
 - Transactions
 - Process monitor

References

- Software Architecture in Practice, 2nd edition by Bass, Clements and Kazman.
- Refactoring for Software Design Smells, Managing Technical Debt – Girish Suryanarayana, Ganesh Samarthym and Tushar Sharma
- Software Architecture – Foundations, Theory and Practice – Richard Taylor, Nenad Medvidovic and Eirc M Dashofy
- Software Architecture, A Case Based Approach – Vasudeva Varma
- Geroge Fairbanks, Rhino Consulting
- The Open Group
- SEI CMU
- Many more !!!

Why the Vasa Sank: 10 Problems and Some Antidotes for Software Projects

Richard E. Fairley, *Oregon Health and Sciences University*

Mary Jane Willshire, *University of Portland*

On 10 August 1628, the Royal Swedish Navy's newest ship set sail on its maiden voyage. The Vasa sailed about 1,300 meters and then, in a light gust of wind, capsized in Stockholm's harbor, losing 53 lives. The ship was Sweden's most expensive project ever, and a total loss. This was a major national disaster—Sweden was at war with Poland and needed the ship for the war effort. A formal hearing the following month did not determine why the ship sank, and no one was blamed.

In 1628, the Royal Swedish Navy launched the Vasa. After sailing only about 1,300 meters, it sank. The authors review the project's problems, including why the ship was unstable and why it was still launched. They interpret the case in terms of today's large, complex software projects and present some antidotes.

After initial salvage attempts, the ship was largely forgotten until Anders Franzen located it in 1956.¹ In 1961, 333 years after it sank, the Vasa was raised; it was so well preserved that it could float after the gun portals were sealed and water and mud were pumped from it. The sheltered harbor had protected the ship from storms, and the Baltic Sea's low salinity prevented worms from infesting and destroying the wooden vessel. Today it is housed in the Vasa Museum (www.vasamuseet.se), near the site where it foundered.² Figures 1 and 2 show the restored ship and a recreation of its sinking.

Researchers have extensively analyzed the Vasa and examined historical records concerning its construction. It sank, of course, because it was unstable. The reasons it was unstable, and launched when known to be

unstable, are numerous and varied. Although we may never know the exact details surrounding the Vasa, this article depicts our "most probable scenario" based on the extensive and remarkably well-preserved documents of the time, evidence collected during visits to the Vasa Museum, information from the referenced Web sites, and publications by those who have investigated the circumstances of the Vasa's sinking. The problems encountered are as relevant to our modern-day attempts to build large, complex software systems as they were to the 17th-century art and craft of building warships.

The Vasa story

The story of the Vasa unfolds as follows.

Changing the shipbuilding orders

On 16 January 1625, Sweden's King Gus-

tav II Adolf directed Admiral Fleming to sign a contract with the Stockholm ship builders Henrik and Arend Hybertsson to design and oversee construction of four ships. Henrik was the master shipwright, Arend the business manager. They subcontracted with shipbuilder Johan Isbrandsson to build the ships under their direction over a period of four years. Two smaller ships were to have about 108-foot keels; the two larger ones, about 135-foot keels.

Based on a series of ongoing and confusing changes the King ordered during the spring and summer of 1625, Henrik requested oak timbers be cut from the King's forest for two 108-foot ships and one 135-foot ship. On 20 September, the Swedish Navy lost 10 ships in a devastating storm. The king then ordered that the two smaller ships be built first on an accelerated schedule to replace two of the lost ships. Construction of the Vasa began in early 1626 as a small, traditional ship; it was completed two and a half years later as a large, innovative ship, after undergoing numerous changes in requirements.

On 30 November 1625, the King changed his order, requiring the two smaller ships to be 120 feet long so that they could carry more armament: 32 24-pound guns in a traditional enclosed-deck configuration.¹ ("24 pounds" refers to the weight of the shot fired by the cannon. A 24-pounder weighed approximately 3,000 pounds.) Henrik inventoried materials and found that he had enough timber to build one 111-foot ship and one 135-foot ship. Under the King's direction, as conveyed by Admiral Fleming, Henrik laid the keel for a 111-foot ship because it could be completed more quickly than the larger one. The records are unclear as to whether the keel was initially laid for this size or initially for a 108-foot ship and then extended to 111 feet.

No specifications for the modified keel

After the Vasa's 111-foot keel had been laid, King Gustav learned that Denmark was building a large ship with two gun decks. The King then ordered the Vasa to be enlarged to 135 feet and include two enclosed gun decks (see Figure 3). No one in Sweden, including Henrik Hybertsson, had ever built a ship with two enclosed gun decks. Because of schedule pressure, the shipbuilders thought that scaling up the



Figure 1. The restored Vasa, housed in Stockholm's Vasa Museum (model in foreground, Vasa in background).

111-foot keel using materials planned for the bigger ship would be more expeditious than laying a new 135-foot keel.

The evolution of warship architecture from one to two enclosed gun decks in the early 1600s marked a change in warfare tactics that became commonplace in the late 1600s and 1700s. The objective became to fire broadside volleys and sink the opponent. Before that, warships fired initial cannon volleys to cripple their opponent's ship so that they could board and seize it. To this end, earlier warships carried large numbers of soldiers (as many as 300).

Although the contract with the Hybertssons was revised (and has been preserved), no one has ever found specifications or crude sketches for either the 111-foot or 135-foot Vasa, and none of the related (and well-preserved) documents mention such drawings. It is unlikely that anyone spent time preparing specifications, given the circumstances and schedule pressure under which the Vasa was constructed. They probably would not have been prepared for



Figure 2. A recreation of the Vasa disaster. (photo by Hans Hammarskiöld)

the original 108-foot ship, because these types of ships had been routinely built for many years and Hybertsson was an experienced shipwright, working with an experienced shipbuilder. Henrik Hybertsson probably “scaled up” the dimensions of the original 108-foot ship to meet the length and breadth requirements of the 111-foot ship and then scaled those up for the 135-foot version of the Vasa.

How he scaled up the 111-foot Vasa to

Figure 3. The Vasa’s two gun decks.



135 feet was constrained by its existing keel, which contained the traditional three scurfs. He added a fourth scarf to lengthen the keel, but the resulting keel is thin in relation to its length, and its depth is quite shallow for a ship of this size.

Hybertsson's assistant, Hein Jacobsson, later said that the Vasa was built one foot, five inches wider than originally planned to accommodate the two gun decks. However, the keel was already laid, so they could make that change in width only in the upper parts of the ship. This raised the center of gravity and contributed to the Vasa's instability (sailing ships are extremely sensitive to the location of the center of gravity; a few centimeters can make a large difference). Also, those outfitting the ship for its first voyage found that the shallow keel did not provide enough space in the hold for the amount of ballast needed to stabilize a 135-foot ship. The keel's thinness required extra bracing timbers in the hold, further restricting the space available for ballast.

Shifting armaments requirements

The numbers and types of armaments to be carried by the scaled-up Vasa went through a number of revisions. Initially, the 111-foot ship was to carry 32 24-pound guns. Then, the 135-foot version was to carry 36 24-pound guns, 24 12-pound guns, eight 48-pound mortars, and 10 smaller guns. After a series of further revisions, the Vasa was to carry 30 24-pounders on the lower deck and 30 12-pounders on the upper deck. Finally, the King ordered that the Vasa carry 64 24-pound guns—32 on each deck—plus several smaller guns (some documents state the required number as 60 24-pound guns).

Mounting only 24-pound guns had the advantage of providing more firepower and allowed standardization on one kind of ammunition, gun carriage, powder charge, and other fittings. However, the upper deck had to carry the added weight of the 24-pound guns in cramped space that had been built for 12-pound guns, further raising the ship's center of gravity. In the end, the Vasa was launched with 48 24-pound guns (half on each deck), because the gun supplier's manufacturing problems prevented delivery of more guns on schedule. Waiting for the additional guns would have interfered with the requirement

Figure 4. The ship's ornate, gilded carvings.

to launch the ship as soon as possible. Another indication of excessive schedule pressure is that the gun castings were of poor quality. They might well have malfunctioned (exploded) during a naval battle.

Artisan shipbuilders constructed the Vasa's rigging and outfitting without explicit specifications or plans, in the traditional manner that had evolved over many years. The King ordered that the ship be outfitted with hundreds of ornate, gilded, and painted carvings depicting Biblical, mythical, and historical themes (see Figure 4). The Vasa was meant to impress by outdoing the Danish ship being built; no cost was spared, making the Vasa the most expensive ship of its time. However, the heavy oak carvings raised the center of gravity and further increased instability.

The shipwright's death

Henrik Hybertsson became seriously ill in 1626 and died in 1627, one year before the Vasa was completed. During the year of his illness, he shared project supervision with Jacobsson and Isbrandsson, but according to historical records, project management was weak. Division of responsibility was not clear and communication was poor; because there were no detailed specifications, schedule milestones, or work plans, it was difficult for Jacobsson to understand and implement Hybertsson's undocumented plans. Communication among Hybertsson, Jacobsson, and Isbrandsson was poor. This resulted in further delays in completing the ship.

Admiral Fleming made Jacobsson (Hybertsson's assistant) responsible for completing the project after Hybertsson's death. At this time and during the subsequent year, 400 people in five different groups worked on the hull, carvings, rigging, armaments, and ballasting—apparently with little, if any, communication or coordination among them. This was the largest work force ever engaged in a single project in Sweden up to that time. There is no evidence that Jacobsson prepared any documented plans after becoming responsible for completing the ship.

No way to calculate stability, stiffness, or sailing characteristics

Methods for calculating the center of gravity, heeling characteristics, and stability



factors for sailing ships were unknown, so ships' captains had to learn their vessels' operational characteristics by trial-and-error testing. The Vasa was the most spectacular, but certainly not the only, ship to capsize during the 17th and 18th centuries.

Measurements taken and calculations performed since 1961 indicate that the Vasa was so unstable that it would have capsized at a heeling over of 10 degrees; it could not have withstood the estimated wind gust of 8 knots (9 miles per hour) that caused the ship to capsize.³ Recent calculations indicate the ship would have capsized in a breeze of 4 knots.

That the wind was so light during the Vasa's initial (and final) cruise is verified by the fact that the crew had to extend the sails by hand upon launch. Lieutenant Petter Gierdsson testified at the formal inquiry held in September 1628: "The weather was not strong enough to pull out the sheets, although the blocks were well lubricated. Therefore, they had to push the sheets out, and one man was enough to hold a sheet."⁴

During the formal inquiry, several witnesses commented that the Vasa was "heavier above than below," but no one pursued the questions of how and why the Vasa had become top-heavy. No one mentioned the weight of the second deck, the guns, the carvings, or other equipment. In those days, most people (including the experts) thought that the higher and more impressive a warship, and the more and bigger the guns it carried, the more indestructible it would be.

Table I
Ten software project problems and some antidotes

Problem area	Antidotes
1. Excessive schedule pressure	Objective estimates More resources Better resources Prioritized requirements Descoped requirements Phased releases
2. Changing needs	Iterative development Change control/baseline management
3. Lack of technical specifications	Development of initial specifications Event-driven updating of specifications Baseline management of specifications A designated software architect
4. Lack of a documented project plan	Development of an initial plan Periodic and event-driven updating Baseline management of the project plan A designated project manager
5. & 6. Excessive and secondary innovations	Baseline control Impact analysis Continuous risk management A designated software architect
7. Requirements creep	Initial requirements baseline Baseline management Risk management A designated software architect
8. Lack of scientific methods	Prototyping Incremental development Technical performance measurement
9. Ignoring the obvious	Back-of-the-envelope calculations Assimilation of lessons learned
10. Unethical behavior	Ethical work environments and work cultures Personal adherence to a code of ethics

The failed prelaunch stability test

Captain Hannson (the ship's captain) and a skeleton crew conducted a stability test in the presence of Admiral Fleming during outfitting of the Vasa. The "lurch" test consisted of having 30 men run from side to side amidship. After three traversals by the men, the test was halted because the ship was rocking so violently it was obvious it would capsize if the test were not halted. The ship could not be stabilized because there was no room to add ballast under the hold's floorboards (see Figure 5). In any case, the additional weight would have placed the lower-deck gun portals near or below the ship's waterline. The Vasa was estimated to be carrying about 120 tons of ballast; it would have needed more than twice that amount to stabilize.

That the Vasa was launched with known stability problems was the result of poor com-

munication, pressure from King Gustav to launch the ship as soon as possible, the fact that the King was in Poland conducting a war campaign (and thus unavailable for consultation), and the fact that no one had any suggestions for making the ship more stable.

Testimony at the formal hearing indicated that Jacobsson, the shipwright, and Isbrandsson, the shipbuilder, were not present during the stability test and were unaware of the outcome. The boatswain, Mattson, testified that Admiral Fleming had accused him of carrying too much ballast, noting that "the gunports are too close to the water!" Mattson then claimed to have answered, "God grant that the ship will stand upright on her keel." To which the Admiral replied, "The shipbuilder has built ships before and you should not be worried."⁴

Whether Admiral Fleming and Captain Hannson intentionally withheld the stability test results is a matter for speculation. The King had ordered that the Vasa be ready by 25 July and "if not, those responsible would be subject to His Majesty's disgrace." The Vasa's maiden voyage on 10 August was more than two weeks later. It was reported that after the failed stability test, Admiral Fleming lamented, "If only the King were here."⁵

Ten problem areas and their antidotes

Many of the Vasa project's problems sound familiar to those who have grappled with large software projects. Table 1 presents 10 problems from the Vasa project that are also common to software projects, along with some antidotes for software projects.

1. Excessive schedule pressure

Many software projects, like the Vasa project, are under excessive schedule pressure to meet a real or imagined need. According to Fred Brooks, more software projects have failed ("gone awry") for lack of adequate calendar time than for all other reasons combined.⁶ Excessive schedule pressure in software projects is caused by

- Truly pressing needs
- Perceived needs that are not truly pressing
- Changing of requirements without adjusting the schedule or the resources (see Problem 2)

- Unrealistic schedules imposed on projects by outside forces
- Lack of realistic estimates based on objective data

When schedule estimates are not based on objective data, software engineers have no basis for defending their estimates or for resisting imposed schedules. You can sometimes meet schedule requirements by increasing project resources, applying superior resources, descoping the (prioritized) requirements, phasing releases, or some combination of these antidotes.

2. Changing needs

Some common reasons for changes to software requirements include competitive forces (as in the case of the Vasa), user needs that evolve over time, changes in platform and target technologies, and new insights gained during a project. There are two techniques for coping with changing software requirements:

- Pursuing an iterative development strategy (for example, incremental, evolutionary, agile, or spiral)
- Practicing baseline management

You can use these techniques alone or together. In the case of iterative development, you can accommodate requirements changes based on priority within the constraints of time, resources, and technology—any of which you can adjust as the project evolves. When using baseline management, you initially place the requirements under version control. Approved changes result in a new version of the requirements (a new baseline), which is accompanied by adjustments to schedule, resources, technology, and other factors as appropriate. The goal of both approaches is to maintain, at all times, a balance among requirements, schedule, resources, technology, and other factors that might pose risks to successful delivery of an acceptable product within the project constraints.

3. Lack of technical specifications

Software projects, like the Vasa project, sometimes start as small, familiar activities for which technical specifications are deemed unnecessary. These projects often



Figure 5. A cross section of the Vasa showing its shallow keel and its multiple decks, including the two gun decks.

grow to become large, innovative ones. In the Vasa's case, verbal communication might have compensated somewhat for the lack of written specifications and a written project plan. In the case of software projects, initially developing and baselining requirements, even for a small, simple project, is much easier (and less risky) than trying to "reverse-engineer" requirements later, when the project has crossed a complexity threshold that warrants developing technical specifications.

4. Lack of a documented project plan

Because its designers saw the Vasa as a small, familiar ship and because the project team was experienced in building these types of ships, they might have thought systematic planning was an unnecessary use of time and resources. Software projects often start under similar circumstances. As in the case of requirements, evolving a baselined project plan for an initially small project is much easier than trying to construct a project plan later (when there is no time to do so). A software project plan must include

- Decomposition of the work to be done (that is, a *work breakdown structure*)
- Allocation of requirements to the work elements of the WBS
- An allocated schedule (for example, a milestone chart)
- Allocation of resources to each schedule increment
- Deliverable work products for each schedule increment

In modern society, the role of engineering is to provide systems and products that enhance the material aspects of human life, thus making life easier, safer, more secure, and more enjoyable.

- Plans for acquiring software from vendors and open sources
- Plans for managing subcontractors
- A risk management plan
- A clear statement of authorities and responsibilities⁷

You can document the project plan for a small software project in a few pages and then expand it as the project grows.

5. Excessive innovation

In *To Engineer Is Human*, Henry Petroski observes that engineering projects often fail when people attempt innovations beyond the state of the art.⁸ Software projects are always innovative to some extent because replicating existing software, unlike replicating physical artifacts, is a trivial process. Software projects are conducted to develop new systems “from scratch” and to produce new versions of existing systems, but not to replicate software. To control excessive innovation in software projects, you can apply the following techniques:

- Baseline control of working documents (for instance, requirements, project plans, design documents, test plans, and source code)
- Impact analysis of proposed changes
- Continuous risk management

6. Secondary innovations

Reasons for secondary innovations in software projects include the need to accommodate the constraints of the technologies used, the addition of derived requirements to support primary requirements and changes to them, and innovations based on the developers’ creative impulses. As in the Vasa’s case, these secondary requirements can overwhelm a software project. In addition to the techniques mentioned for controlling Problem 5, you should assign responsibility and give authority to a designated software architect to maintain the software product’s vision and integrity, which is an additional antidote for many other problems.⁷

7. Requirements creep

It seems that no one was aware of the overall impact of the changes made to the Vasa during the two and a half years of construction. This can (and does) easily happen

to software projects. Fundamental techniques for maintaining control of software projects are developing initial documentation and consistently updating requirements and plans to maintain an acceptable balance among requirements, schedule, and resources as the project evolves. Often-heard excuses for failure to establish initial project documentation are lack of sufficient knowledge to do so and the belief that “everything will change anyway, so why plan?” You should develop initial requirements and plans to the extent possible in the face of imperfect knowledge, with the expectation that they will change and with procedures in place to evolve them systematically. Failure to update initial requirements and plans periodically, and as events require, is often the result of a perception that there is not enough time to do so (“never enough time to do it right, but always time to redo it”). This perception, in turn, comes from insufficient procedures for managing requirements and plans on a continuing basis and a lack of appreciation for the risks thus created.

8. Lack of scientific methods

Because software has no physical properties, you cannot calculate many traditional engineering parameters for software (at least, at this time). Unlike physical artifacts such as the Vasa, however, you can build a software system in small, incremental steps and monitor the evolution of technical parameters such as memory usage, performance, safety, security, and reliability as the system evolves.

9. Ignoring the obvious

In the case of the Vasa, the lurch test demonstrated that the ship was dangerously unstable. There was no room for additional ballast. If there had been room, the added weight would have placed the lower gun portals below the waterline. Although we lack scientific methods in many software engineering areas, we can often avoid foreseeable disasters by performing a few, often quite simple, “back of the envelope” calculations. If, for example, a certain transaction processing system must process 1,000 transactions per second and if each transaction requires four complex queries, the system must process 4,000 complex queries per second, including the time required for con-

About the Authors



Richard E. Fairley is a professor of computer science and associate dean of the OGI School of Science & Engineering of the Oregon Health & Science University. He also participates in the Oregon Master of Software Engineering program, which is offered collaboratively by four Oregon universities. His research interests include requirements engineering, software architecture, software process engineering, estimation, measurement and control of software projects, risk management, and software engineering education. He is a member of the IEEE Computer Society and ACM. Contact him at the OGI School of Science and Eng., 20000 NW Walker Rd., Beaverton, OR 97006; dfairley@cse.ogi.edu.



Mary Jane Willshire is an associate professor of computer science in the Electrical Engineering and Computer Science Department at the University of Portland, where she teaches courses and conducts research in software engineering, human-computer interfaces, and database technology. She is a member of the IEEE, ACM, Association for Women in Science, and Society of Women Engineers. Contact her at the School of Engineering, Univ. of Portland, 5000 N. Willamette Blvd., Portland, OR 97203-5798; willshir@up.edu.

text switching among transactions (that is, 250 microseconds per context switch and transaction). Accounting for system latencies might show the envisioned system to be infeasible.

10. Unethical behavior

In modern society, the role of engineering is to provide systems and products that enhance the material aspects of human life, thus making life easier, safer, more secure, and more enjoyable. Technological innovation often involves ethical considerations. In the Vasa's case, the goal was to make Sweden more secure for its citizens and to bring glory to the country. At the last moment, when it became obvious the ship was not seaworthy, those with authority to stop the launch did not do so.

In our time, software engineers should first serve the public; second, be advocates for the customers and users of their products; and third, act in the interest of their employers, in a manner that is consistent with the public interest. Software engineers should "accept full responsibility for their work" and "approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy, or harm the environment. The ultimate effect of the work should be to the public good."⁹ Every software engineer should be familiar with and adhere to a code of ethics.

On a positive note, large, two-deck warships were subsequently built and sailed during the latter 17th and the 18th and 19th centuries. In the same way, we can now, with reasonable confidence of success, build larger and more complex software-intensive systems than in the past. ☐

Acknowledgment

The reviewers and editor offered many valuable recommendations for improvements to this article's initial draft. Photos by permission of the Vasa Museum.

References

1. A. Franzen, *The Warship Vasa: Deep Diving and Marine Archaeology*, 6th ed., Norstedt & Soners Forlag, Stockholm, 1974.
2. *The Royal Ship Vasa*, <http://home.swipnet.se/~w-70853/WASAe.htm>.
3. C. Borgestam and A. Sandstrom, *Why Vasa Capsized*, AB Grafisk Press, Stockholm, Sweden, 1995.
4. *The Story of the Royal Swedish Man-of-War Vasa*, multi-media documentary, VyKett AB, Stockholm, Sweden, 1999; available (in English) on floppy disk by email inquiry to the Vasa Museum at vasamuseet@sshm.se.
5. A. Wahlgren, *The Warship Vasa*, a documentary film, Vasa Museum, Stockholm, Sweden, 1996; available (in English) on a VCR tape by email inquiry to the Vasa Museum at vasamuseet@sshm.se.
6. F. Brooks, *The Mythical Man-Month*, Addison-Wesley, Boston, 1995.
7. *IEEE Std. 1058-1998, Standard for Software Project Management Plans*, IEEE, Piscataway, N.J., 1998.
8. Henry Petroski, *To Engineer Is Human: The Role of Failure in Successful Design*, Vintage Books, New York, 1992.
9. *Software Engineering Code of Ethics and Professional Practice*, IEEE Computer Society and ACM Joint Task Force on Software Engineering Ethics and Professional Practices, 1999, <http://computer.org/tab/code11.htm>.

Table 1 presents only some of the many possible antidotes for software projects. Some antidotes listed in one problem area apply equally to other problem areas. For the sake of brevity, we leave elaboration of Table 1 to readers.

According to the formal Vasa hearing's transcript, no one asked how or why the ship had become unstable or why it was launched with known stability problems. The failure of this line of inquiry is perhaps the most compelling problem from the Vasa case study, as is our often-observed, present-day failure to learn from our mistakes in software engineering.

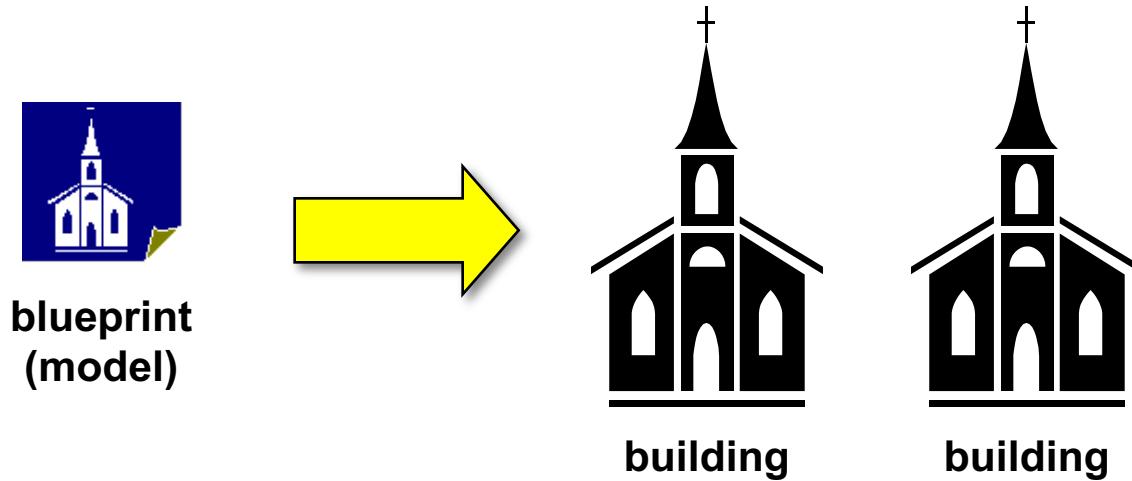
For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

Software Modeling using UML

Software Engineering (Spring 2021)
IIIT Hyderabad

Models

- A *model* is a description of something
 - “*a pattern for something to be made*” (Merriam-Webster)

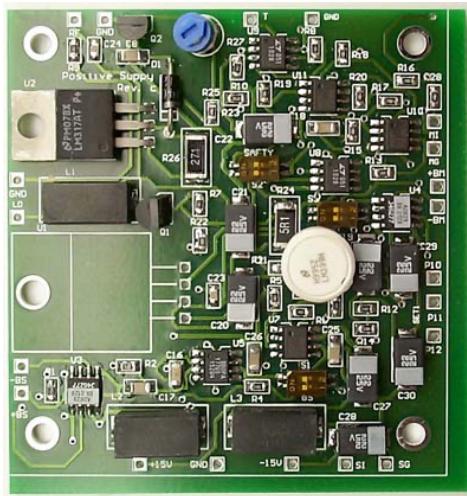


- model ≠ thing that is modeled
 - The Map is Not The Territory

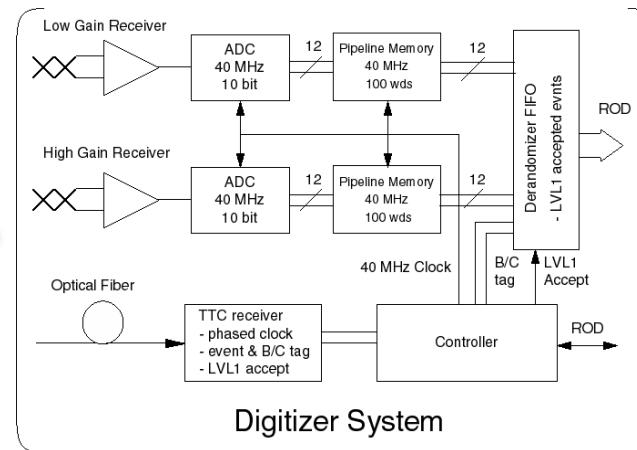
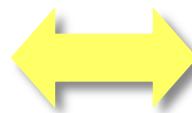
Engineering Models

- Engineering model:

A reduced representation of some system that highlights the properties of interest from a given viewpoint



Modeled system



Functional Model

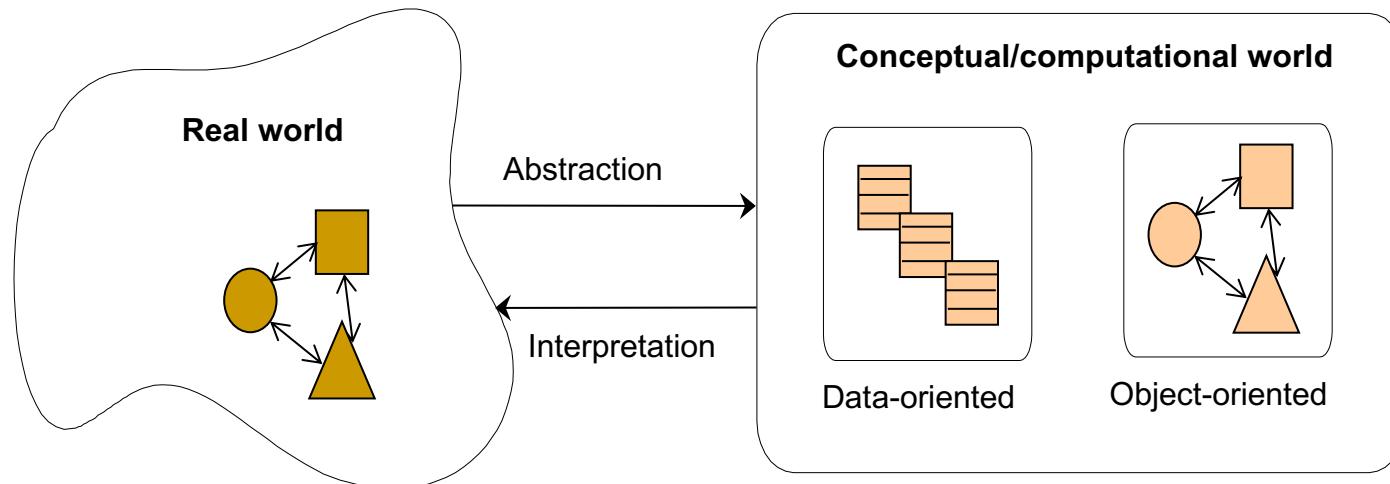
- ◆ We don't see everything at once
- ◆ We use a representation (notation) that is easily understood for the purpose on hand

Models

- How do we model?
- Modeling Maturity Level
 - Level 0: No specification
 - Level 1: Textual
 - Level 2: Text with Diagrams
 - Level 3: Models with Text
 - Level 4: Precise Models

Object-Oriented Modeling

- Uses object-orientation as a basis of modeling
- Models a system as a set of objects that interact with each others
- No semantic gap (or impedance mismatch)
- Seamless development process



Key Ideas of O-O Modeling

- Abstraction
- Encapsulation
- Relationship
 - Association: relationship between objects
 - Inheritance: mechanism to represent similarity among objects
- Object-oriented
 - = object (class) + inheritance + message send

Objects vs. Classes

	Interpretation in the Real World	Representation in the Model
Object	An <i>object</i> represents anything in the real world that can be distinctly identified.	An <i>object</i> has an identity, a state, and a behavior.
Class	A <i>class</i> represents a set of objects with similar characteristics and behavior. These objects are called <i>instances</i> of the class.	A <i>class</i> characterizes the structure of states and behaviors that are shared by all of its instances.

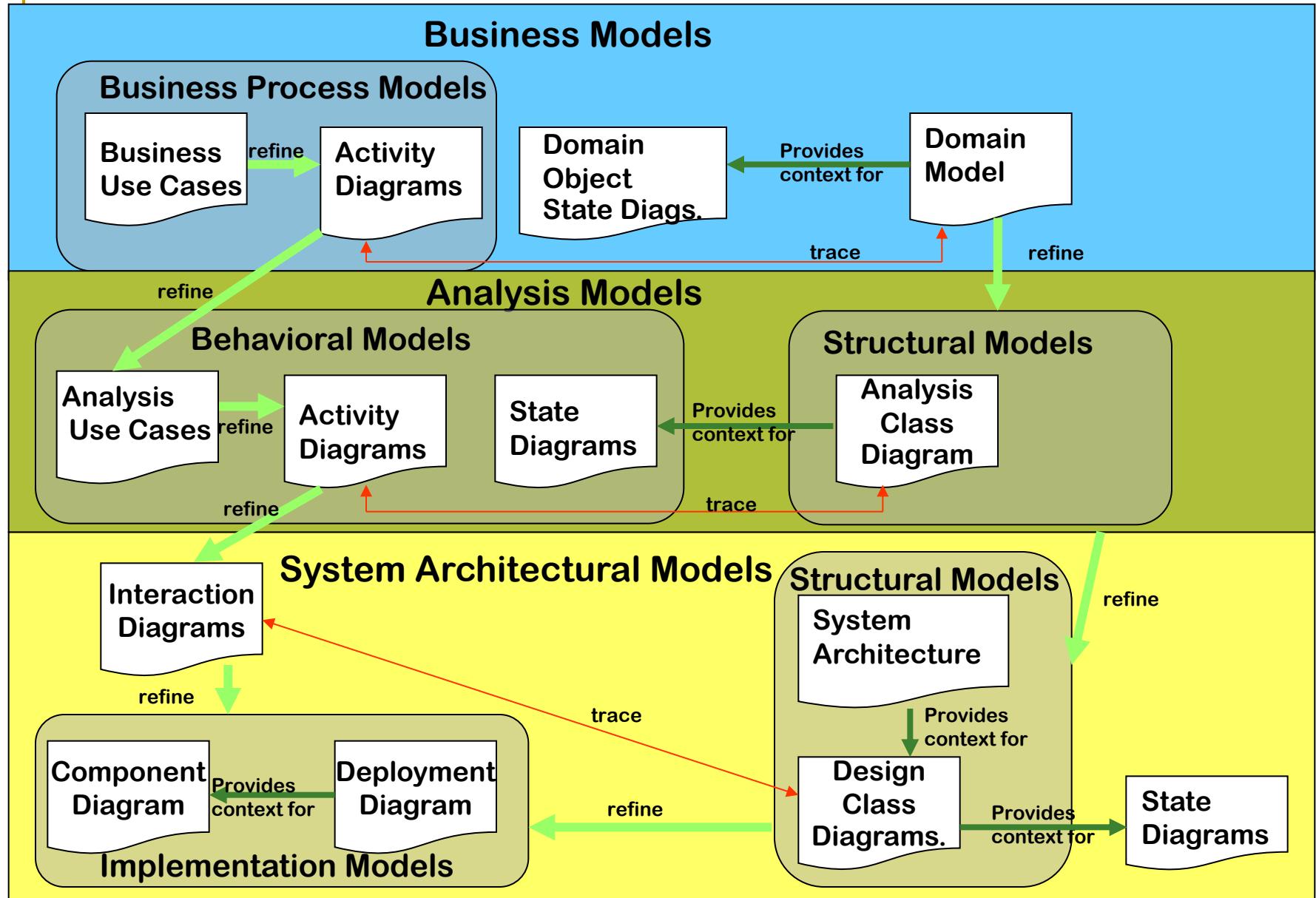
Unified Modeling Language (UML)

- Notation for object-oriented modeling
- Standardized by Object Management Group (OMG)
- Consists of 12+ different diagrams
 - Use case diagram
 - Class diagram
 - Statechart diagram
 - Sequence diagram
 - Communication diagram
 - Component diagram
 - ...

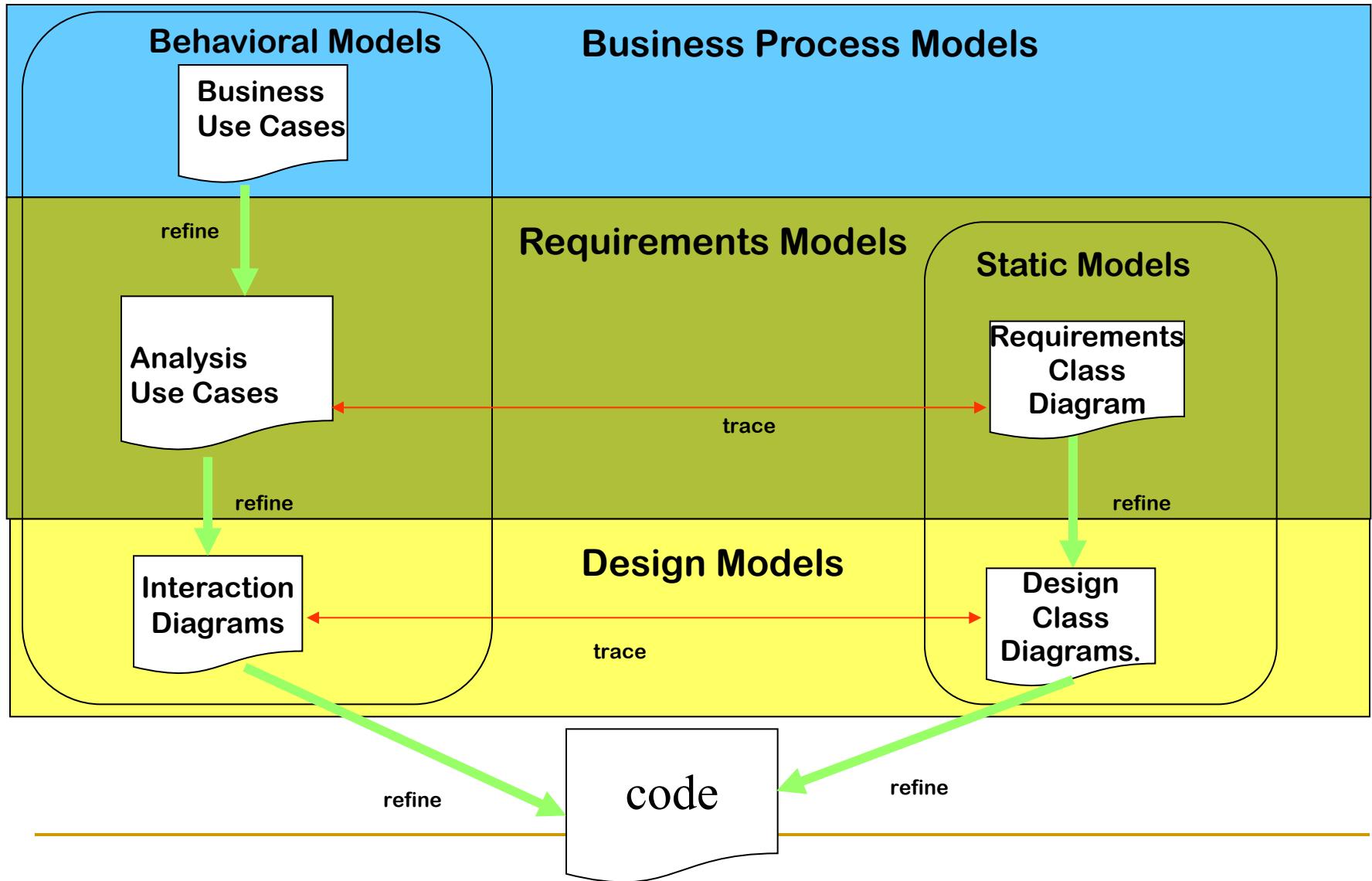
What the UML is not

- Not an OO method or process
- Not a visual programming language
- Not a tool specification

A “Full” Process (using UML diagrams)



An “UltraLite” Process

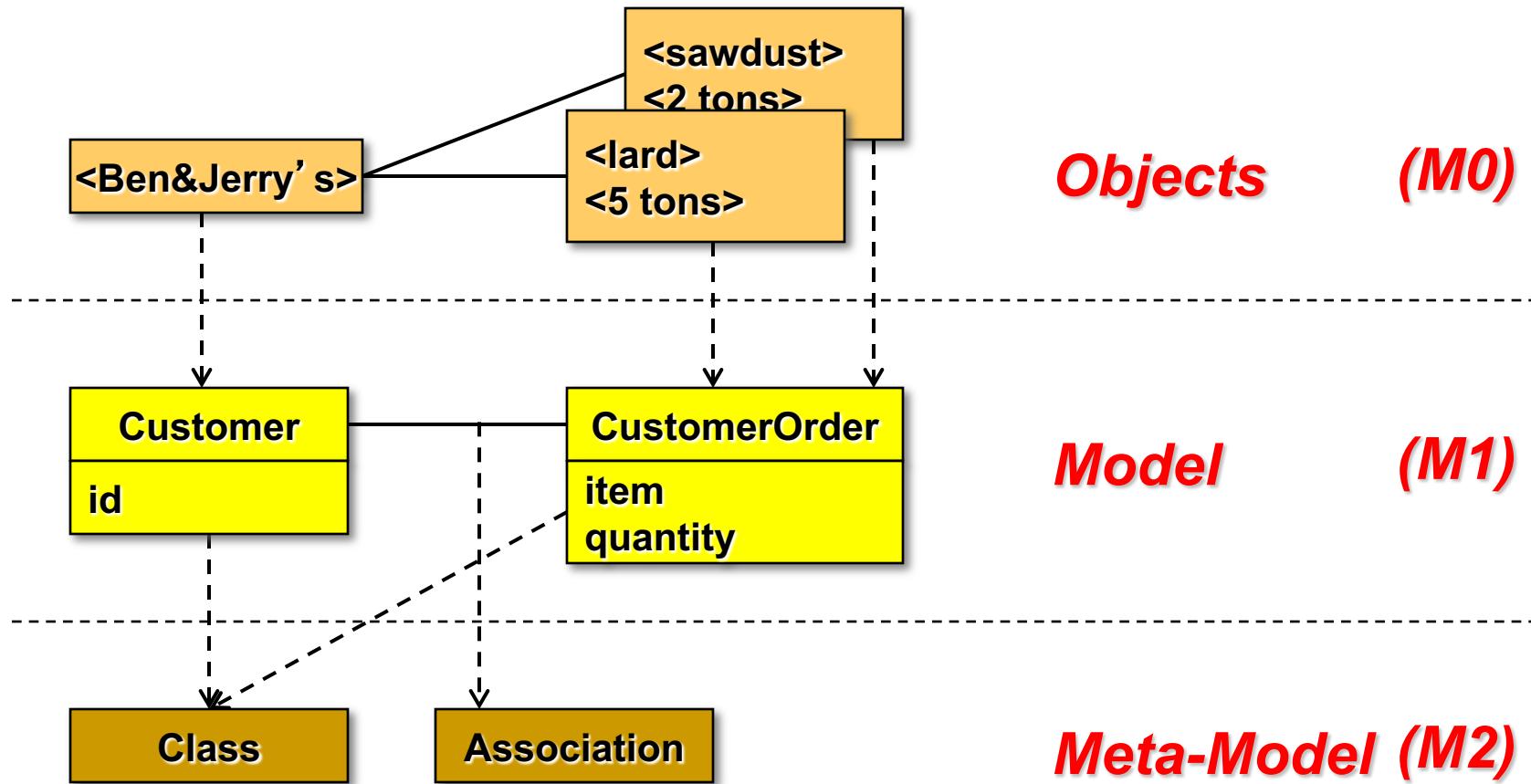


Static vs. Dynamic Models

- Static model
 - Describes static structure of a system
 - Consists of a set of objects (classes) and their relationships
 - Represented as class diagrams
- Dynamic model
 - Describes dynamic behavior of a system, such as state transitions and interactions (message sends)
 - Represented as statechart diagram, sequence diagrams, and collaboration diagrams

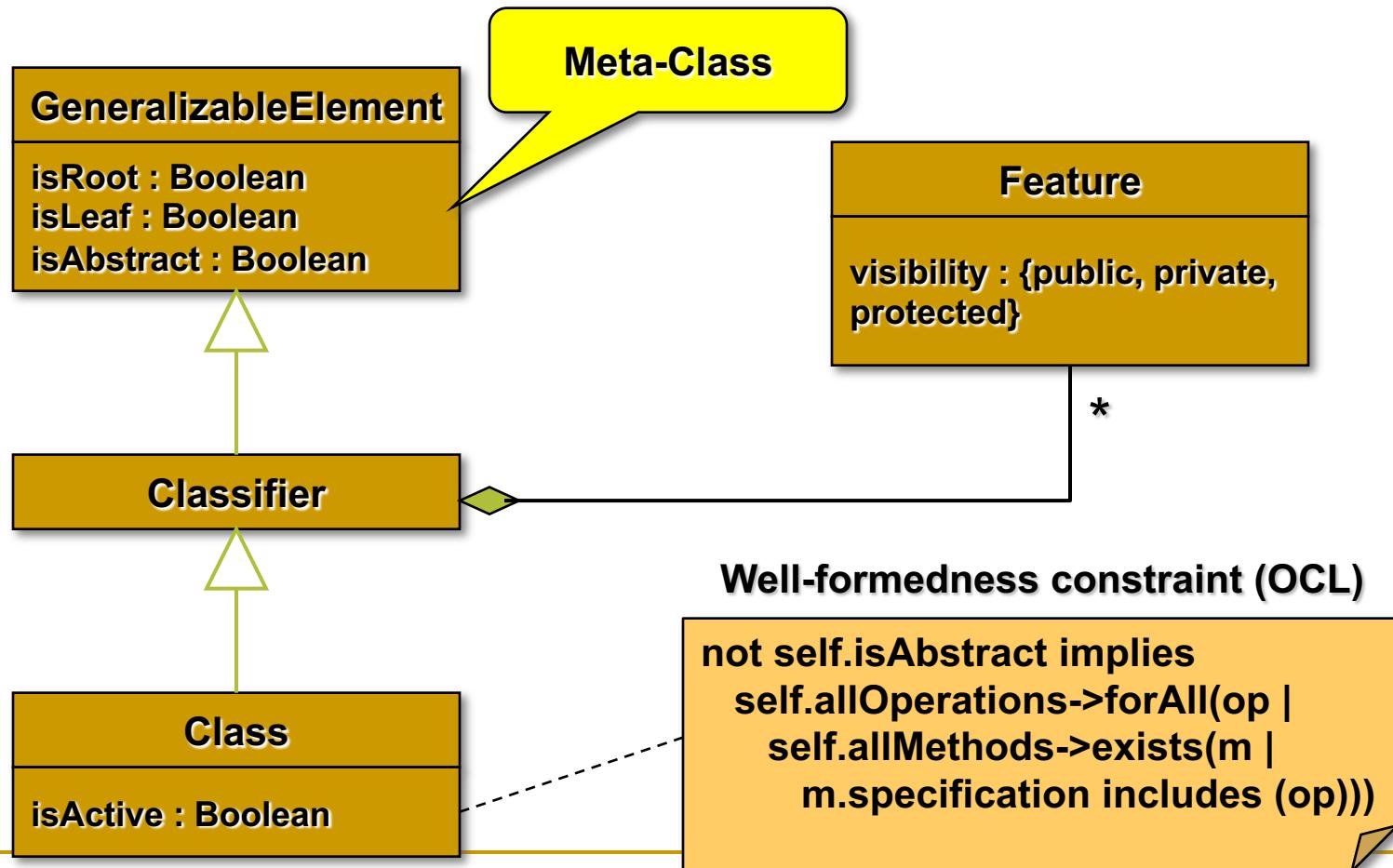
Models and Meta-Models

- Meta-models are simply Models of Models



The UML Meta-Model

- Is a UML Model of UML



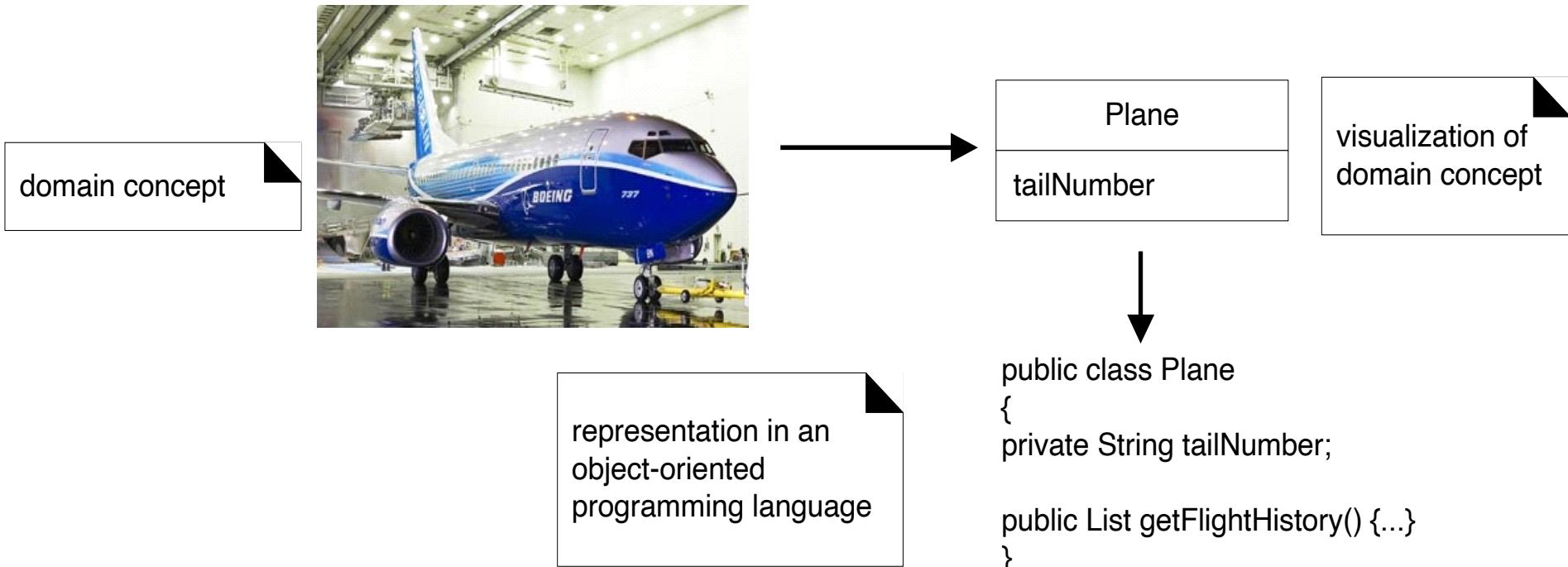
Domain concept

vs.

design representation of domain concept

vs.

code representation of domain concept

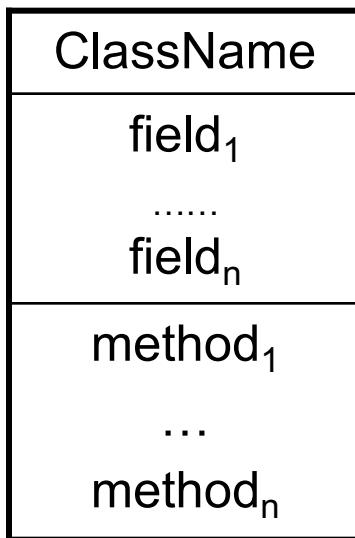


UML Class Diagram

- Most common diagram in OO modeling
- Describes the static structure of a system
- Consist of:
 - Nodes representing classes
 - Links representing of relationships among classes
 - Inheritance
 - Association, including aggregation and composition
 - Dependency

Notation for Classes

- The UML notation for classes is a rectangular box with as many as three compartments.

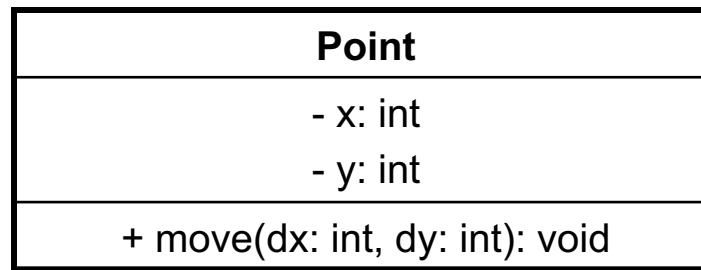
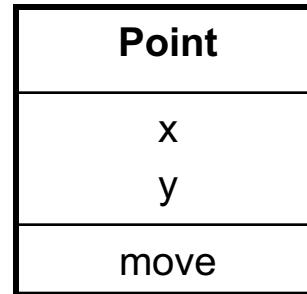


The top compartment show the class name.

The middle compartment contains the declarations of the fields, or *attributes*, of the class.

The bottom compartment contains the declarations of the methods of the class.

Example



Field and Method Declarations in UML

- Field declarations
 - birthday: Date
 - +duration: int = 100
 - -students[1..MAX_SIZE]: Student
- Method declarations
 - +move(dx: int, dy: int): void
 - +getSize(): int

Visibility	Notation
public	+
protected	#
package	~
private	-

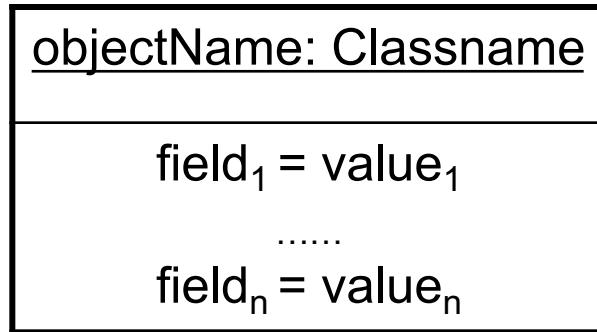
Exercise

- Draw a UML class diagram for the following Java code.

```
class Person {  
    private String name;  
    private Date birthday;  
    public String getName() {  
        // ...  
    }  
    public Date getBirthday() {  
        // ...  
    }  
}
```

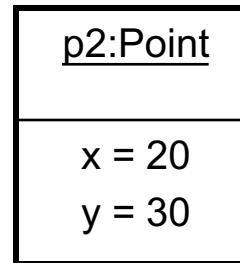
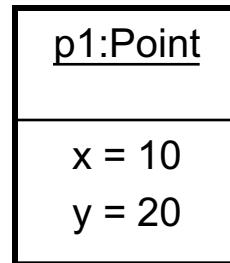
Notation for Objects

- Rectangular box with one or two compartments



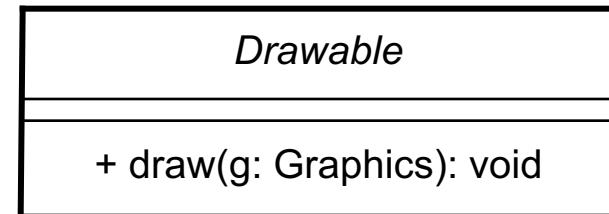
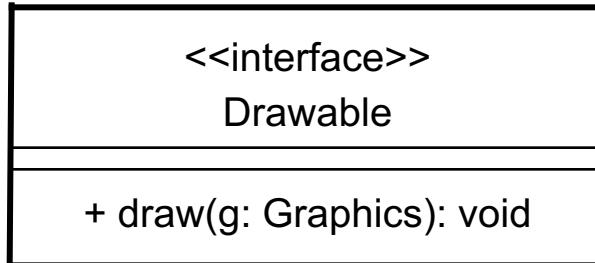
The top compartment shows the name of the object and its class.

The bottom compartment contains a list of the fields and their values.



UML Notation for Interfaces

```
interface Drawable {  
    void draw(Graphics g);  
}
```

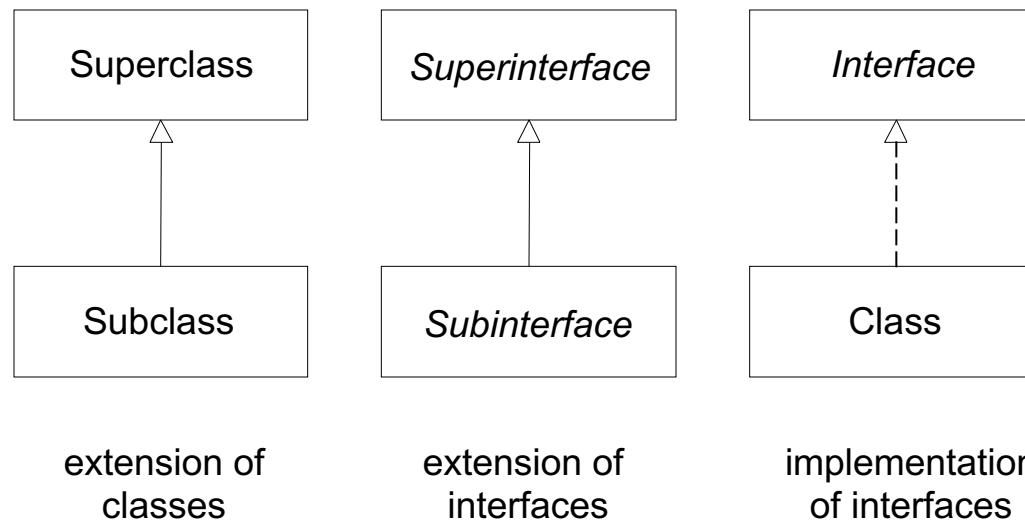


Inheritance in Java

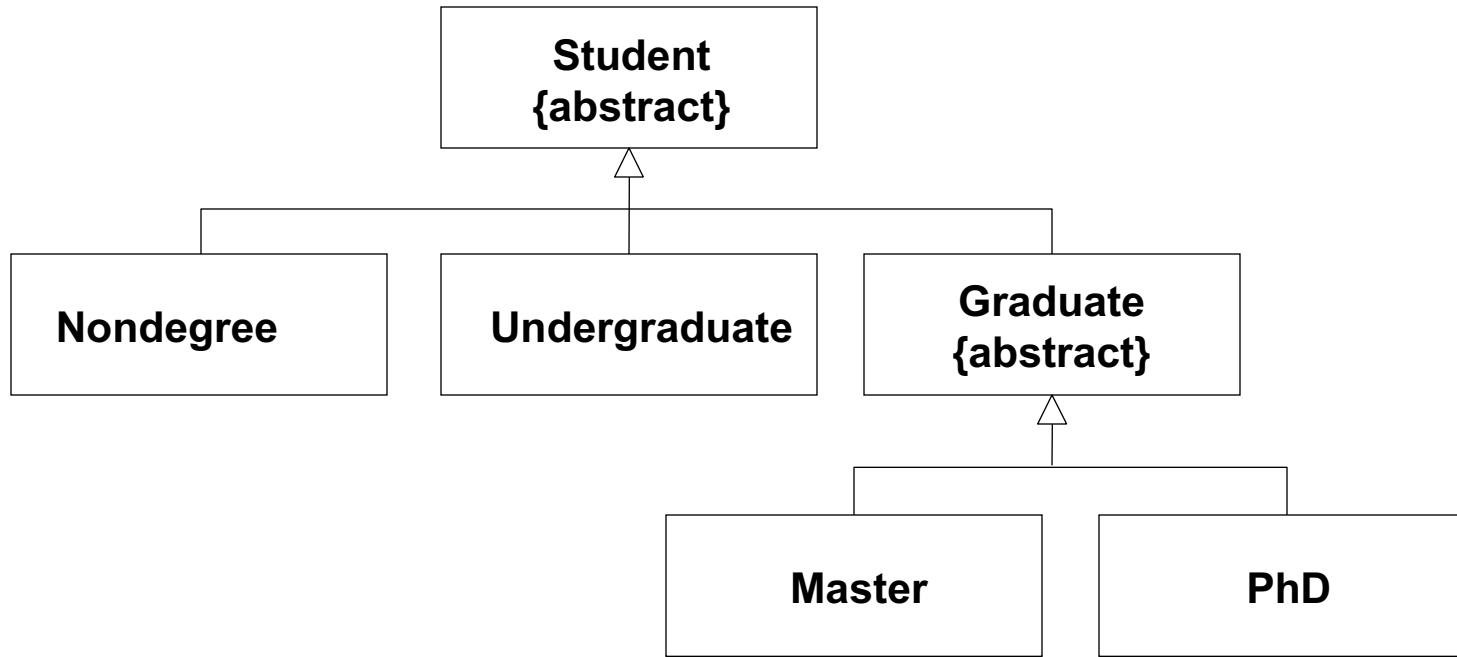
- Important relationship in OO modeling
- Defines a relationship among classes and interfaces.
- Three kinds of inheritances
 - *extension* relation between two classes (*subclasses* and *superclasses*)
 - *extension* relation between two interfaces (*subinterfaces* and *superinterfaces*)
 - *implementation* relation between a class and an interface

Inheritance in UML

- An extension relation is called *specialization* and *generalization*.
- An implementation relation is called *realization*.



Example

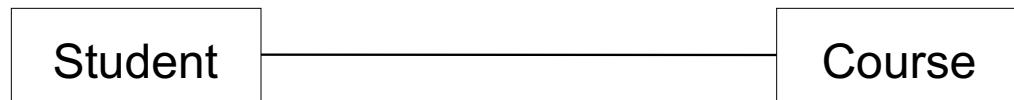


Exercise

- Draw a UML class diagram showing possible inheritance relationships among classes Person, Employee, and Manager.

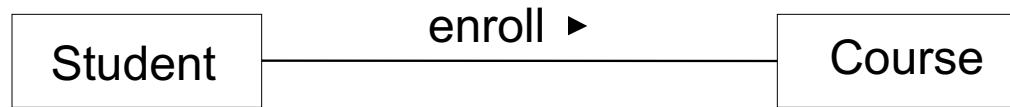
Association

- General binary relationships between classes
- Commonly represented as direct or indirect references between classes



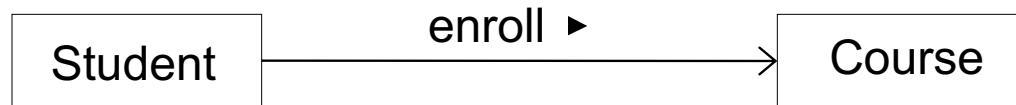
Association (Cont.)

- May have an optional label consisting of a name and a direction drawn as a solid arrowhead with no tail.
- The direction arrow indicates the direction of association with respect to the name.



Association (Cont.)

- An arrow may be attached to the end of path to indicate that *navigation* is supported in that direction
- What if omitted?

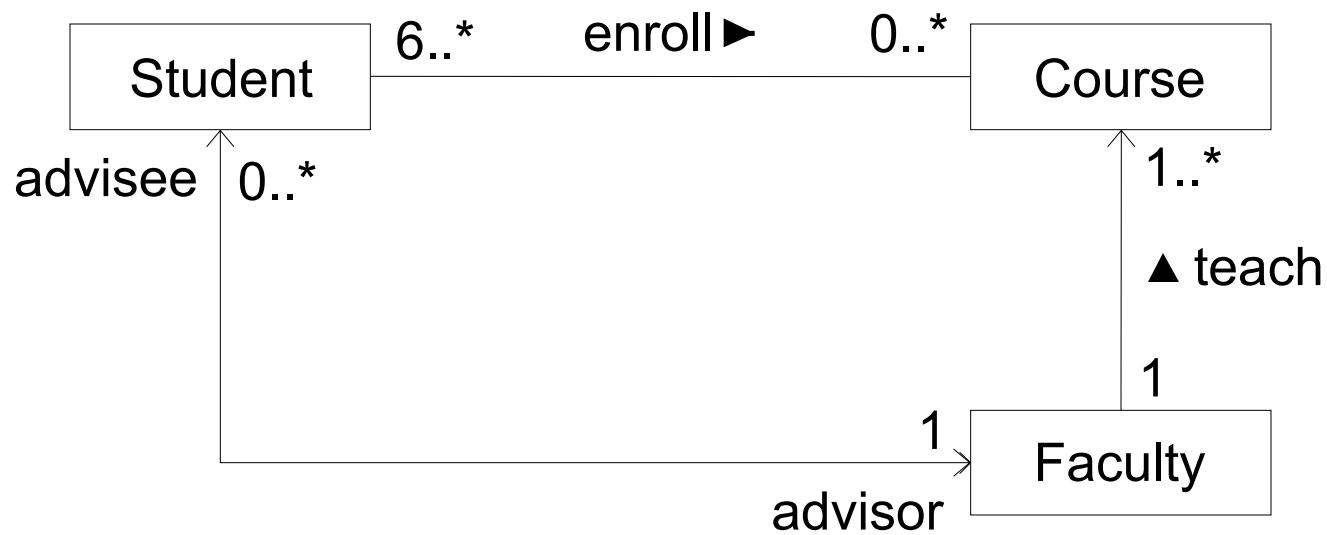


Association (Cont.)

- May have an optional *role name* and an optional *multiplicity specification*.
- The multiplicity specifies an integer interval, e.g.,
 - $l..u$ closed (inclusive) range of integers
 - i singleton range
 - $0..^*$ entire nonnegative integer, i.e., 0, 1, 2, ...



Example

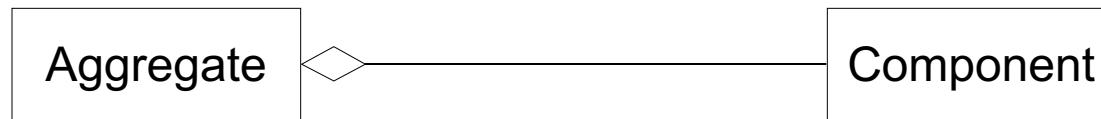


Exercise

- Identify possible relationships among the following classes and draw a class diagram
 - Employee
 - Manager
 - Department

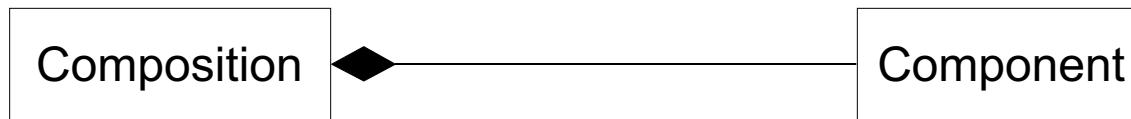
Aggregation

- Special form of association representing *has-a* or *part-whole* relationship.
- Distinguishes the whole (aggregate class) from its parts (component class).
- No relationship in the lifetime of the aggregate and the components (can exist separately).

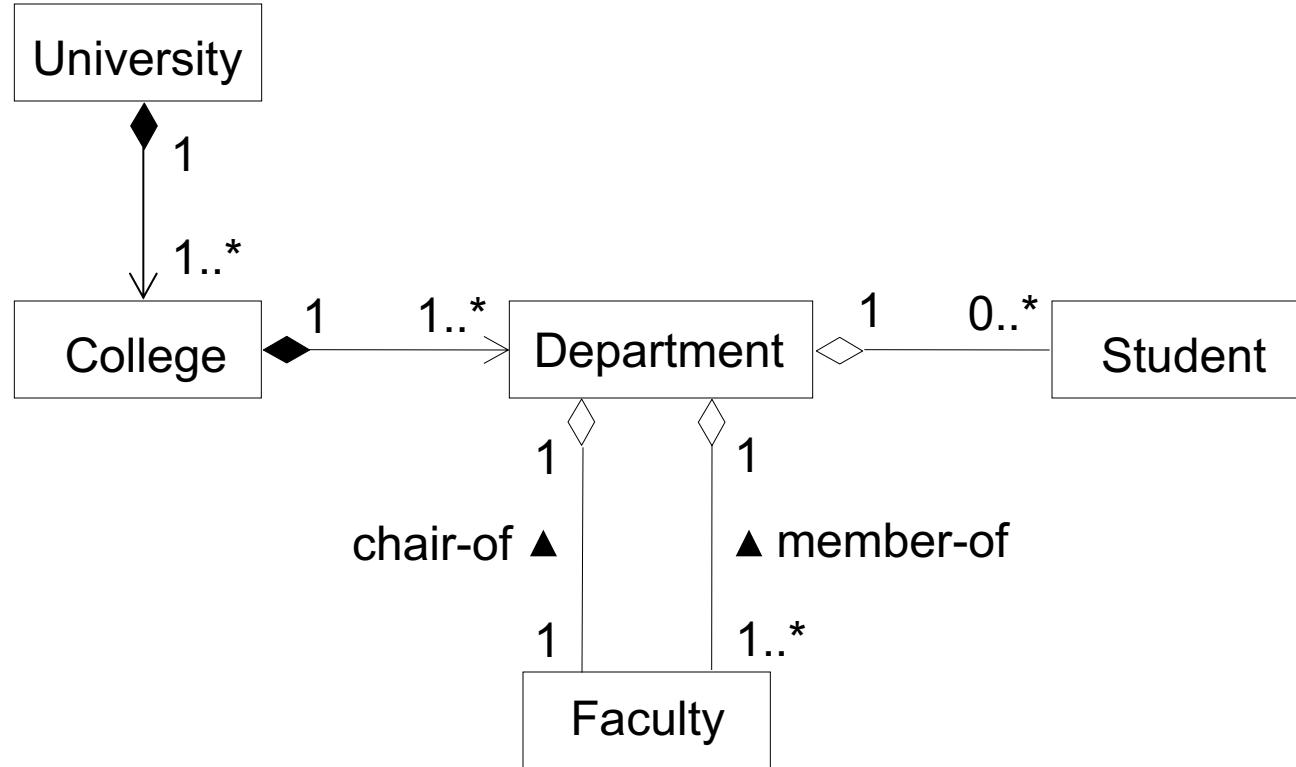


Composition

- Stronger form of aggregation
- Implies exclusive ownership of the component class by the aggregate class
- The lifetime of the components is entirely included in the lifetime of the aggregate (a component can not exist without its aggregate).



Example

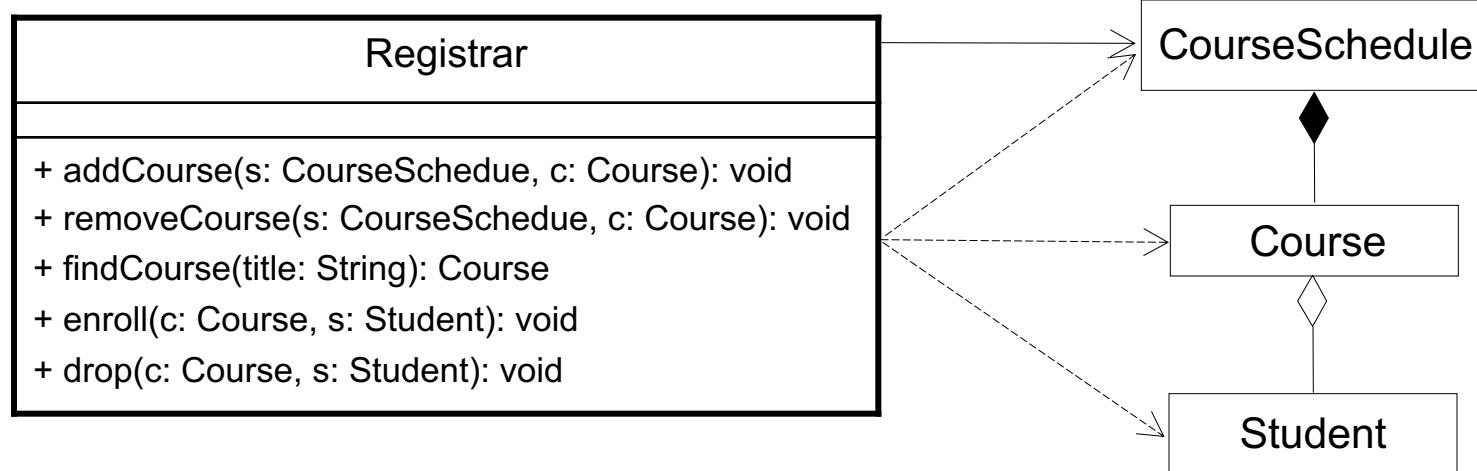


Dependency

- Relationship between the entities such that the proper operation of one entity depends on the presence of the other entity, and changes in one entity would affect the other entity.
- The common form of dependency is the *use* relation among classes.



Example



Dependencies are most often omitted from the diagram unless they convey some significant information.

Group Exercise: E-book Store

Develop an OO model for an e-bookstore. The core requirements of the e-bookstore are to allow its customers to browse and order books, music CDs, and computer software through the Internet. The main functionalities of the system are to provide information about the titles it carries to help customers make purchasing decisions; handle customer registration, order processing, and shipping; and support management of the system, such as adding, deleting, and updating titles and customer information.

1. Identify classes. Classes can represent physical objects, people, organizations places, events, or concepts. Class names should be noun phrases.
2. Identify relevant fields and methods of the classes. Actions are modeled as the methods of classes. Method names should be verb phrases.
3. Identify any inheritance relationships among the classes and draw the class diagram representing inheritance relationships.
4. Identify any association relationships among the classes and draw the class diagram representing association relationships.
5. Identify any aggregation and composition relationships among the classes and draw the class diagram representing dependency relationships.

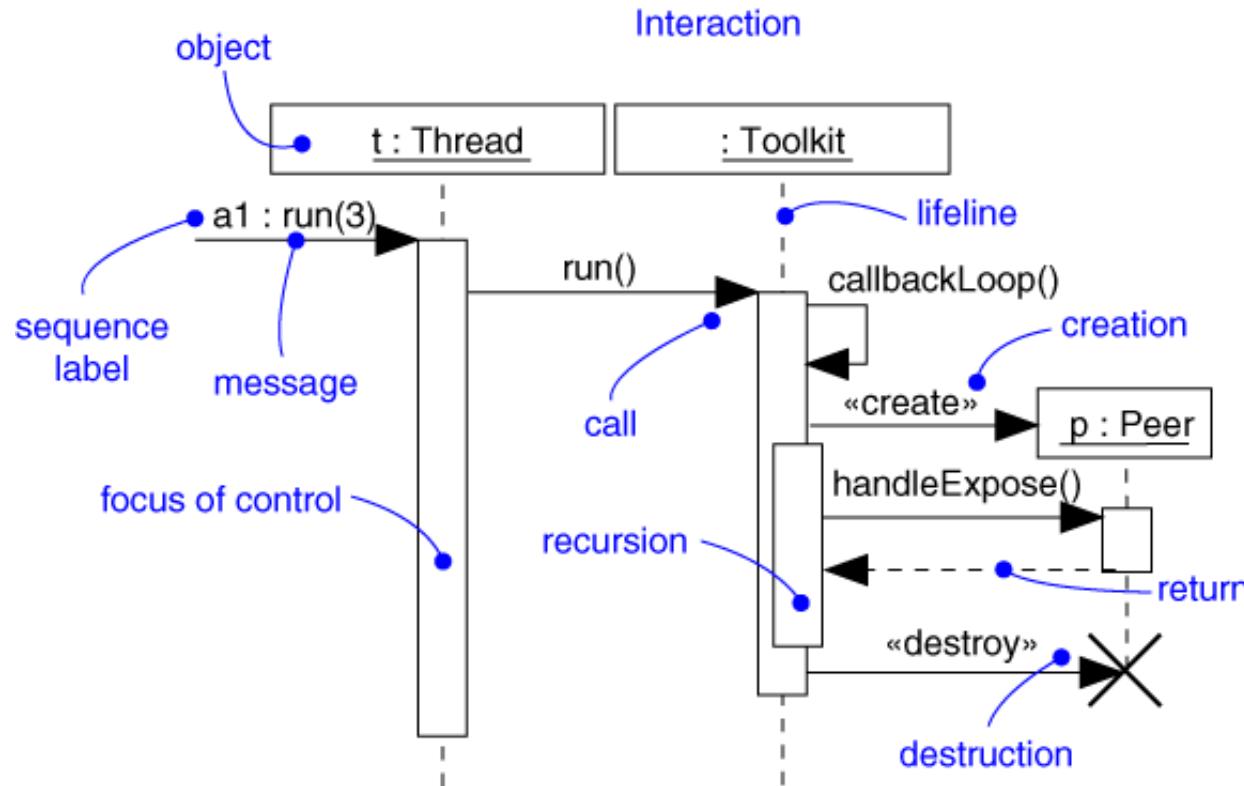
Interaction models

UML Interaction models

- An interaction model shows the interactions that take place between objects in a system
- An interaction “is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose” (UML user guide)
- Interaction models provide a view of system behavior

Sequence Diagram

- Captures dynamic behavior (time-oriented)
- Purpose
 - Model flow of control
 - Illustrate typical scenarios



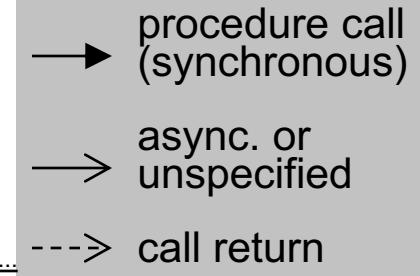
Sequence Diagram Notation

→ class instances (objects)

↓ time

RegisterForCoursesForm IRegistrationController ICourseData...

: Student



object lifeline

message or stimulus

call to self

activation
(focus of control)

- the time during which the instance is performing the action (directly or through a sub-action)

implied return
(the call return arrow is optional)

new instance

destroyed instance

1. createSchedule() >

1.1. getCourseOfferings() >

1.1.1. getCourseOfferings(Semester) >

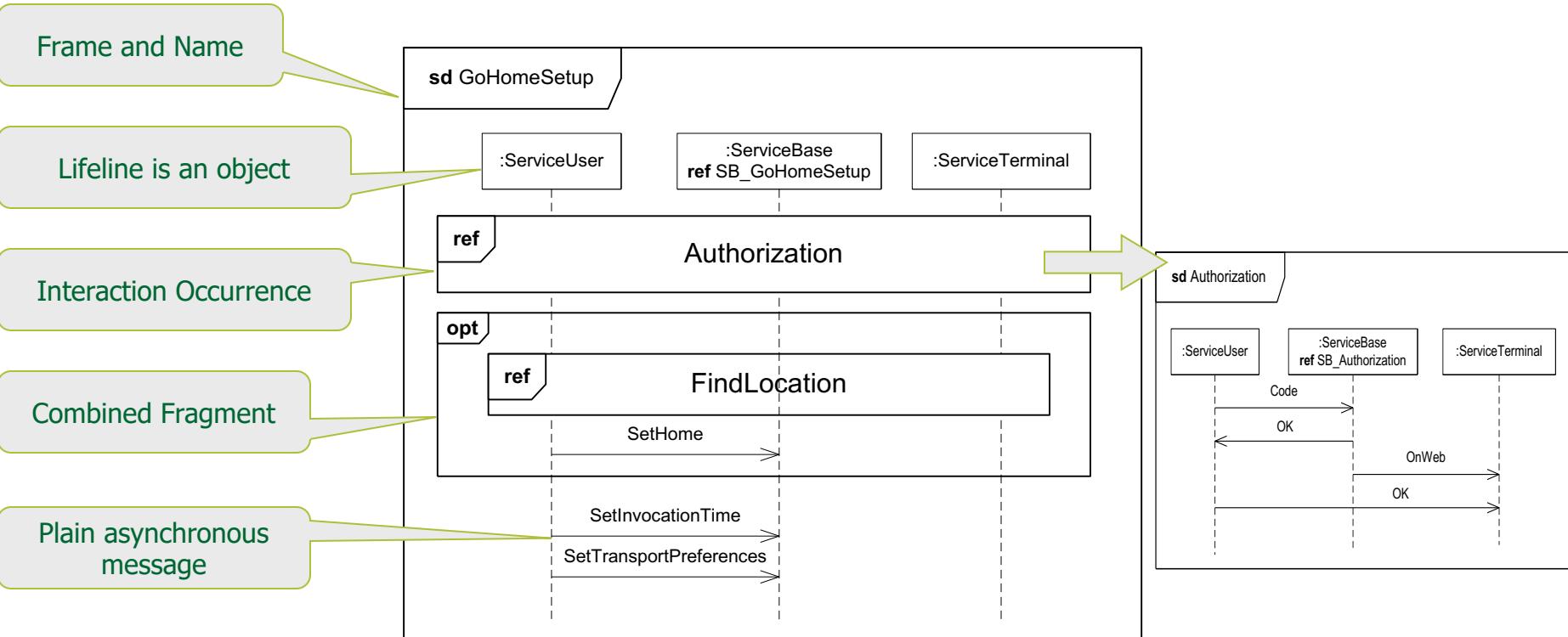
1.2. displayCourseOfferings() <

1.3. new() >

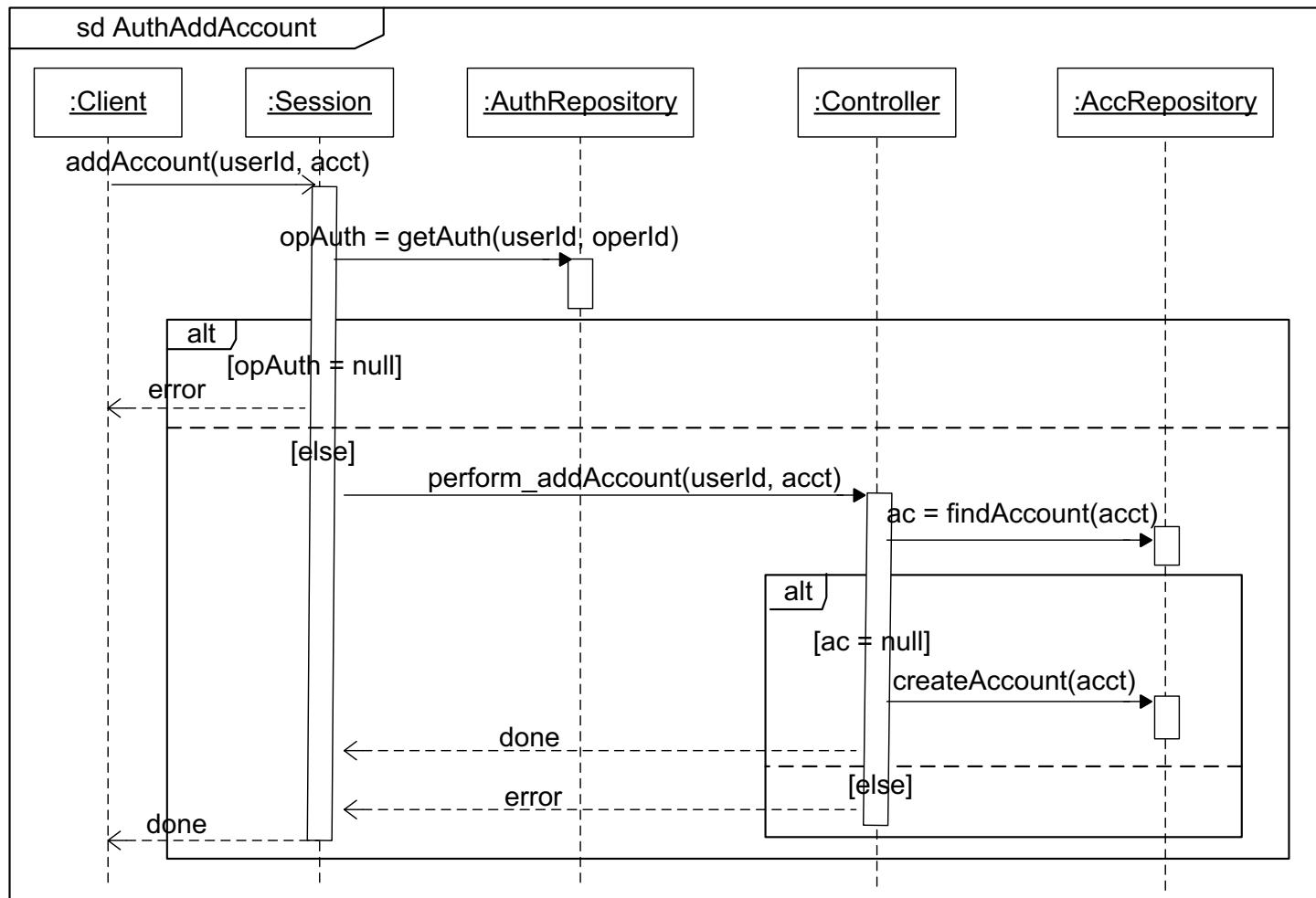
> : Schedule

X

A More Complex Sequence Diagram (UML 2.0)



Sequence diagram - Example



Combined Fragment Types

- Alternatives (**alt**)
 - choice of behaviors – at most one will execute
 - depends on the value of the guard (“else” guard supported)
- Option (**opt**)
 - Special case of alternative
- Break (**break**)
 - Represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop)
- Parallel (**par**)
 - Concurrent (interleaved) sub-scenarios
- Negative (**neg**)
 - Identifies sequences that must not occur

Combined Fragment Types

- Critical Region (**region**)
 - Traces cannot be interleaved with events on any of the participating lifelines
- Assertion (**assert**)
 - Only valid continuation
- Loop (**loop**)
 - Optional guard: [<min>, <max>, <Boolean-expression>]
 - No guard means no specified limit

Different Kinds of Arrows



Procedure call or other kind of nested flow of control

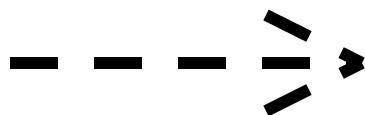
UML 1.4: Asynchronous



Flat flow of control

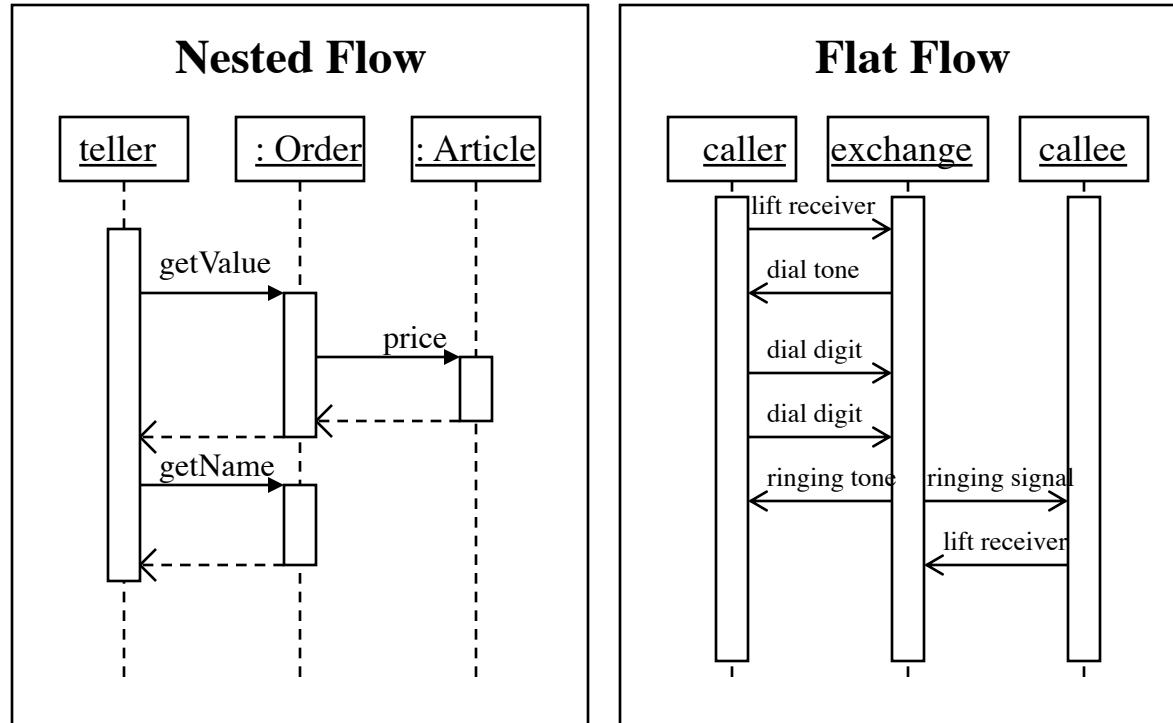


Explicit asynchronous flow of control



UML 1.4: Variant of async
Return

Example: Different Arrows



Interaction Modeling Tips

- Set the context for the interaction.
- Express the flow from left to right and from top to bottom.
- Put active objects to the left/top and passive ones to the right/bottom.
- Use sequence diagrams
 - to show the explicit ordering between the stimuli
 - when modeling real-time

Design principles and practices

Software Engineering
(Spring 2021: Week 3)

IIT Hyderabad

Key design activity

- Assigning responsibilities to classes
 - A *responsibility* is something that the class is required to do (contract or obligation of a class)
 - All the responsibilities of a given class should be *clearly related*.
 - If a class has too many responsibilities, consider *splitting* it into distinct classes
 - If a class has no responsibilities attached to it, then it is probably *useless*
 - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created
- Determine which objects need to know of other objects (determine class navigability)

Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

Why Patterns?

- Design for re-use is difficult
- Experienced designers:
 - Rarely start from first principles
 - Apply a working "handbook" of approaches
- Patterns make this ephemeral knowledge available to all
- Support evaluation of alternatives at higher level of abstraction

Discussion question:

“New Pattern” is an Oxymoron

General Responsibility Assignment Software Patterns or Principles (GRASP)

Information Expert: A general principle of object design and responsibility assignment?

Assign a responsibility to the information expert – the class that has the information necessary to fulfill the responsibility

Creator: Who creates?

Assign class B the responsibility to create an instance of class A if one of these is true:

- (1) B contains A
- (2) B aggregates A
- (3) B has the initializing data for A
- (4) B records A
- (5) B closely uses A

Controller: What first object beyond the UI layer receives and coordinates (“controls”) a system operation?

Assign the responsibility to an object representing one of these choices:

(1) Represents the overall “system”, a “root object”, a device that the software is running within, or a major subsystem (these are all variations of a façade controller)

(2) Represents a use case scenario

GRASP

Low Coupling: How to reduce the impact of change?

Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives

High Cohesion: How to keep objects focused, understandable, and manageable, and as a side-effect, support low coupling?

Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives

Polymorphism: Who is responsible when the behavior varies by type?

When related alternatives or behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies

Pure fabrication: Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

Assign a highly cohesive set of responsibilities to an artificial or convenience “behavior” class that does not represent the problem domain concept – something made up, in order to support high cohesion, low coupling, and reuse.

GRASP

Indirection: How to assign responsibilities to avoid direct coupling?

Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

Protected Variations: How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?

Identify points of predicted variations or instability; assign responsibilities to create a stable “interface” around them

Pattern Types

- **Requirements Patterns:** Characterize families of requirements for a family of applications
 - The checkin-checkout pattern can be used to obtain requirements for library systems, car rental systems, video systems, etc.
- **Architectural Patterns:** Characterize families of architectures
 - The Broker pattern can be used to create distributed systems in which location of resources and services is transparent (e.g., the WWW)
 - Other examples: MVC, Pipe-and-Filter, Multi-Tiered
- **Design Patterns:** Characterize families of low-level design solutions
 - Examples are the popular Gang of Four (GoF) patterns
- **Programming idioms:** Characterize programming language specific solutions

Example of a Software Development Pattern

- The Model-View-Controller (MVC) Pattern

- The *Model* component: encapsulates core functionality; independent of input/output representations and behavior.
- The *View* components: displays data from the model component; there can be multiple views for a single model component.
- The *Controller* components: each view is associated with a controller that handles inputs; the user interacts with the system via the controller components.

The Gang of Four Catalog Method

- Pattern name and classification
 - Purpose classification: creational, structural, behavioral
 - Scope classification: class (compile time) or object (run-time)
- Creational
 - class => defer creation to subclasses
 - object => defer creation to another object
- Structural
 - class => structure via inheritance
 - object => structure via composition
- Behavioral
 - class => algorithms/control via inheritance
 - object => algorithms/control via object groups

Pattern Description - 1

- Intent
 - What does pattern do?
 - What is its rationale?
 - What issue/problem does it address?
- Also Known As = other names for pattern
- Motivation
 - Scenario illustrating problem and solution
 - A concrete exemplar
- Applicability
 - When to apply the pattern
 - Poor designs addressed by pattern

Pattern Description - 2

■ Structure

- Graphical representation – OMT/UML

■ Participants

- Classes, objects, and their responsibilities

■ Collaborations

- Class/object interactions

Pattern Description - 3

- Consequences
 - How does pattern meet its objectives
 - Tradeoffs and results
 - Flexibility: what parts can vary independently?
- Implementation & Sample Code
- Known Uses
- Related Patterns

Refactoring

(Some material adapted from Martin Fowler's book)

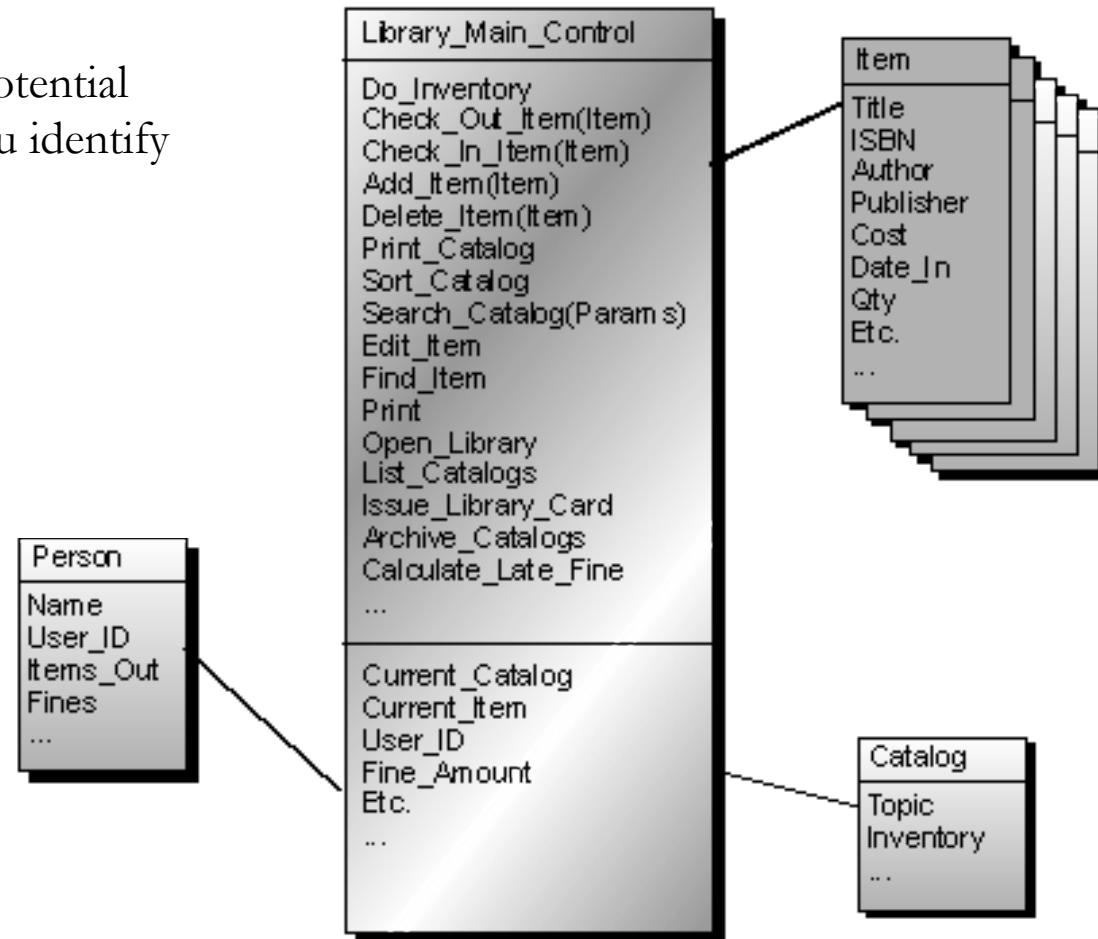
Spring 2021
Software Engineering

Lehman & Belady: Laws of Software Evolution (1974)

- **Continuing Change** - Systems must be continually adapted else they become progressively less satisfactory.
- **Increasing Complexity** - As a system evolves its complexity increases **unless work is done to maintain or reduce it**.

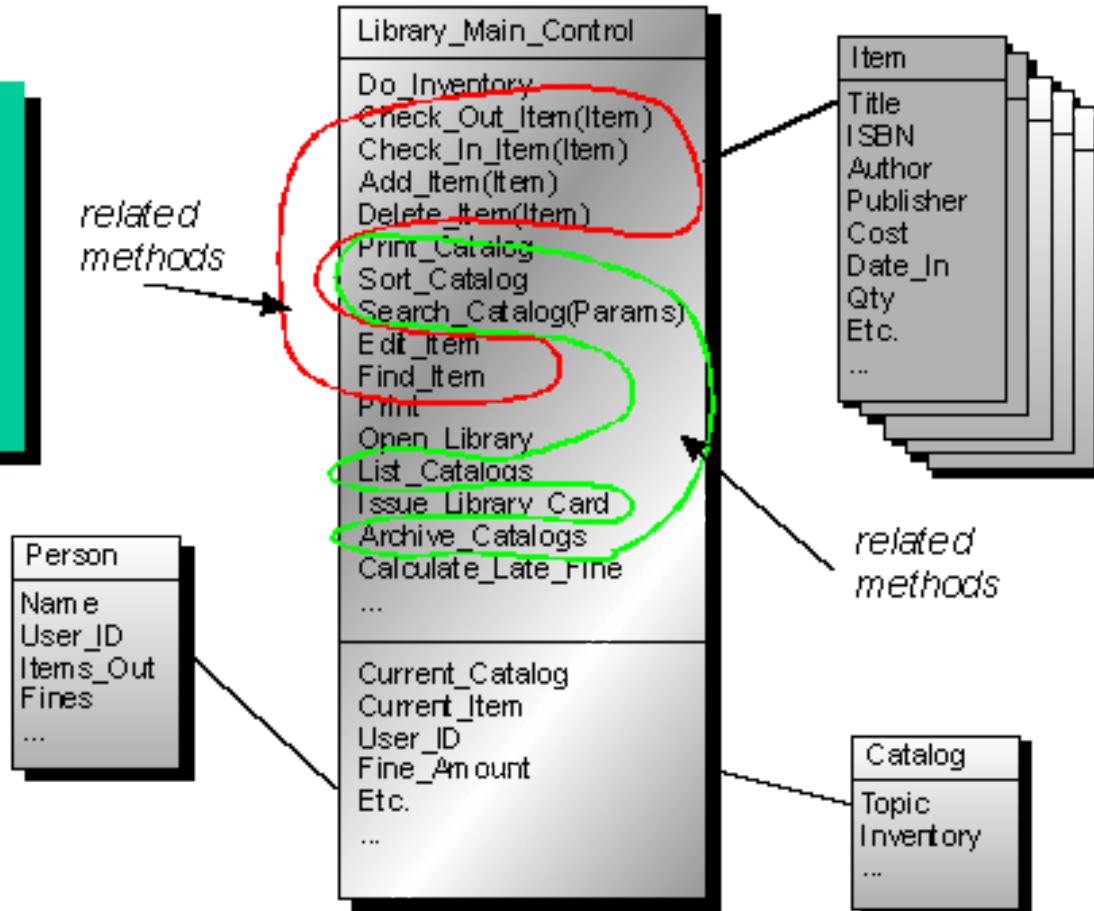
Library system – Existing design

What areas do you see as potential problem areas? Why did you identify each of those areas?



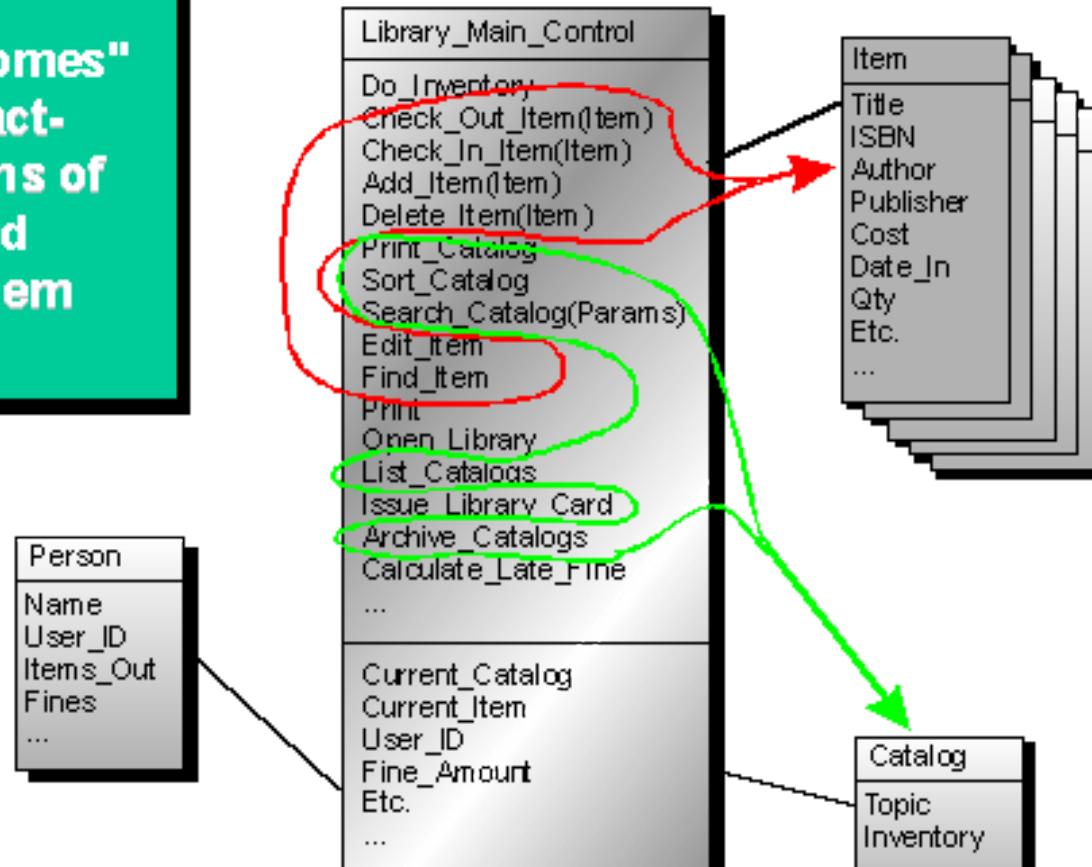
Library system – Changing the design

Step 1:
Identify or categorize related attributes and operations according to contracts.



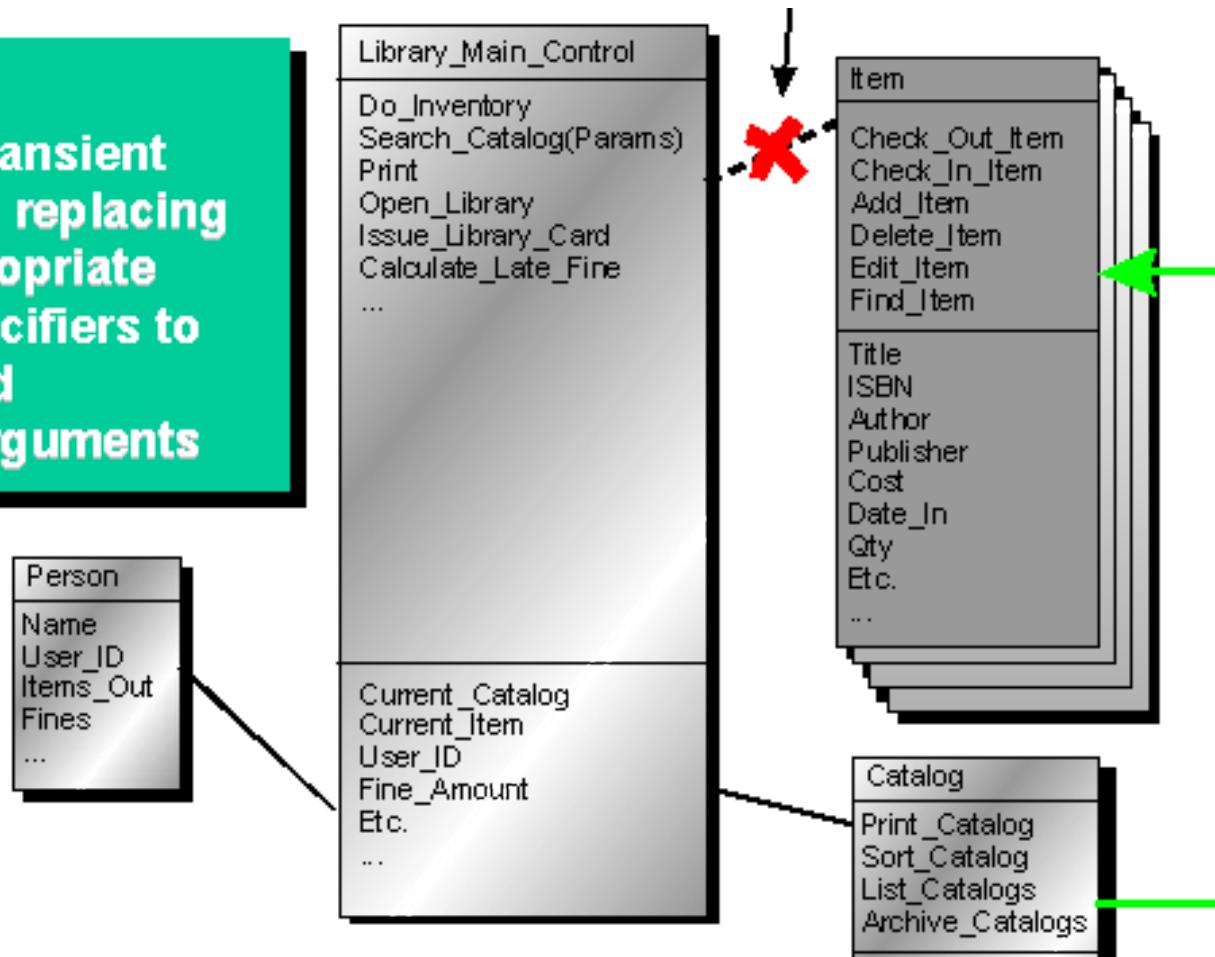
Library system – Changing the design

Step 2:
Find "natural homes"
for these contract-
based collections of
functionality and
then migrate them
there

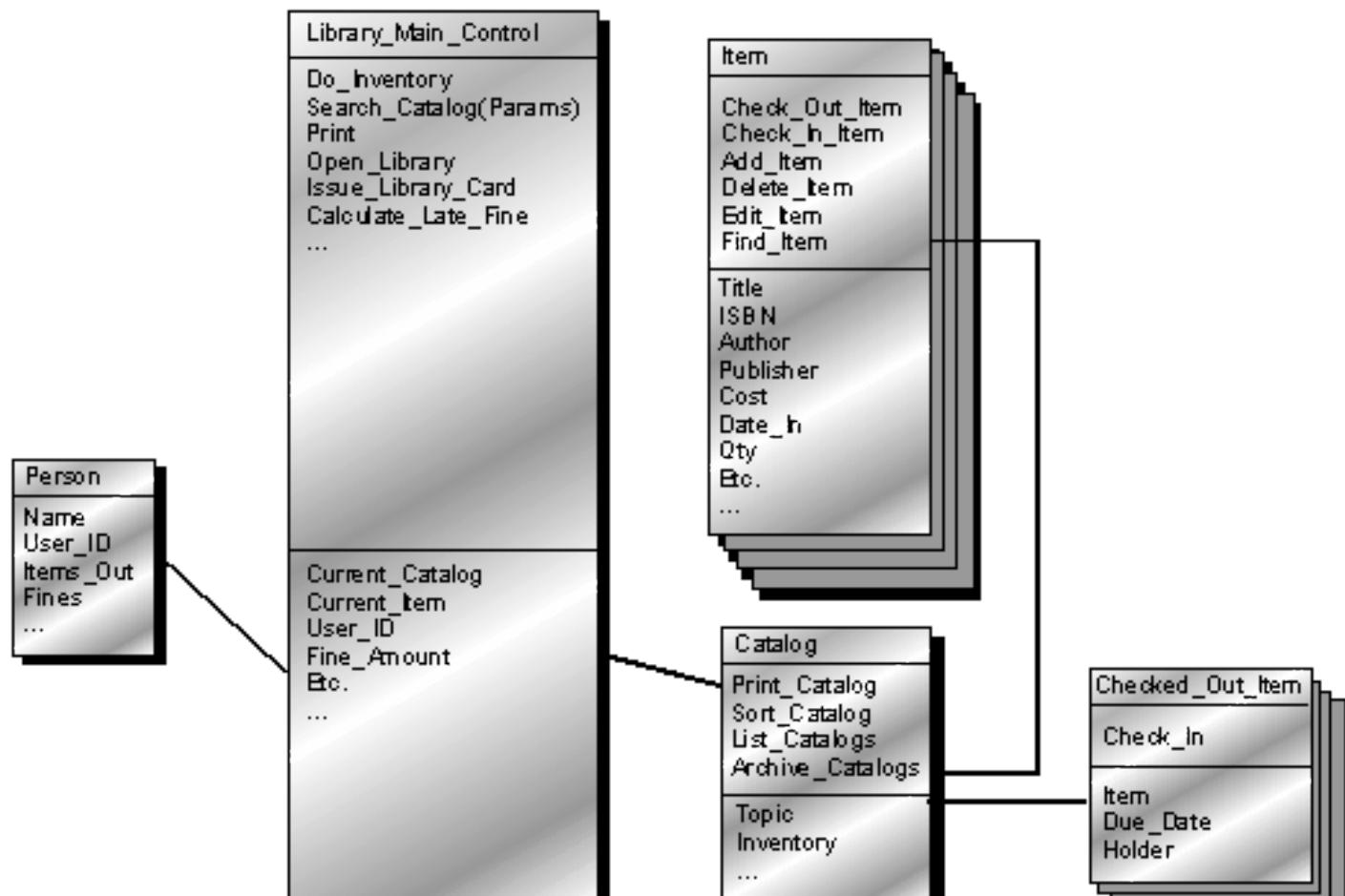


Library system – Changing the design

Final Step:
**Remove all transient
associations, replacing
them as appropriate
with type specifiers to
attributes and
operations arguments**



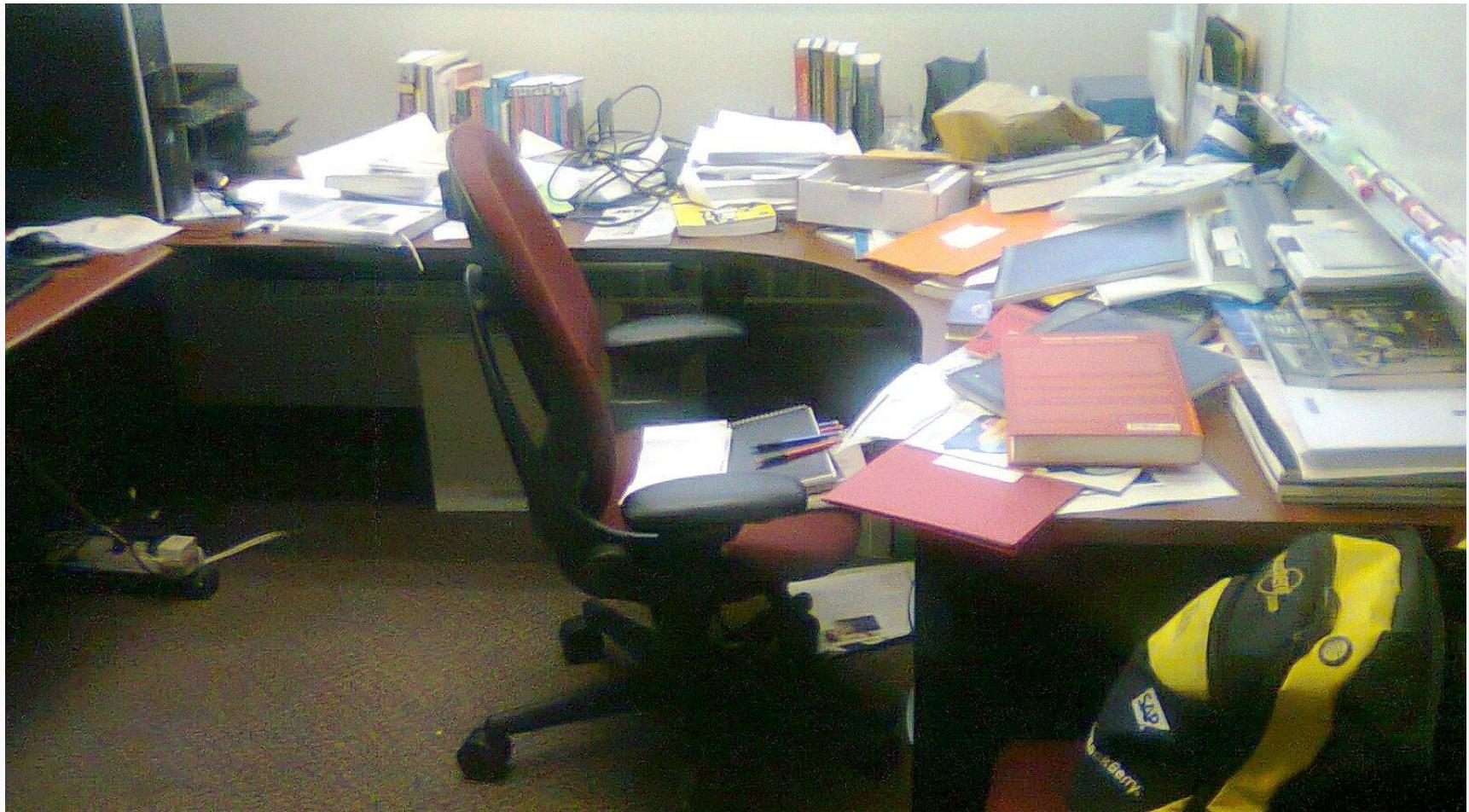
Library system – Changing the design



It is usually hard to counter, “If it ain’t broke, don’t fix it.”

- Generally improves product quality
- Pay today to ease work tomorrow
- May actually accelerate today’s work

From this...



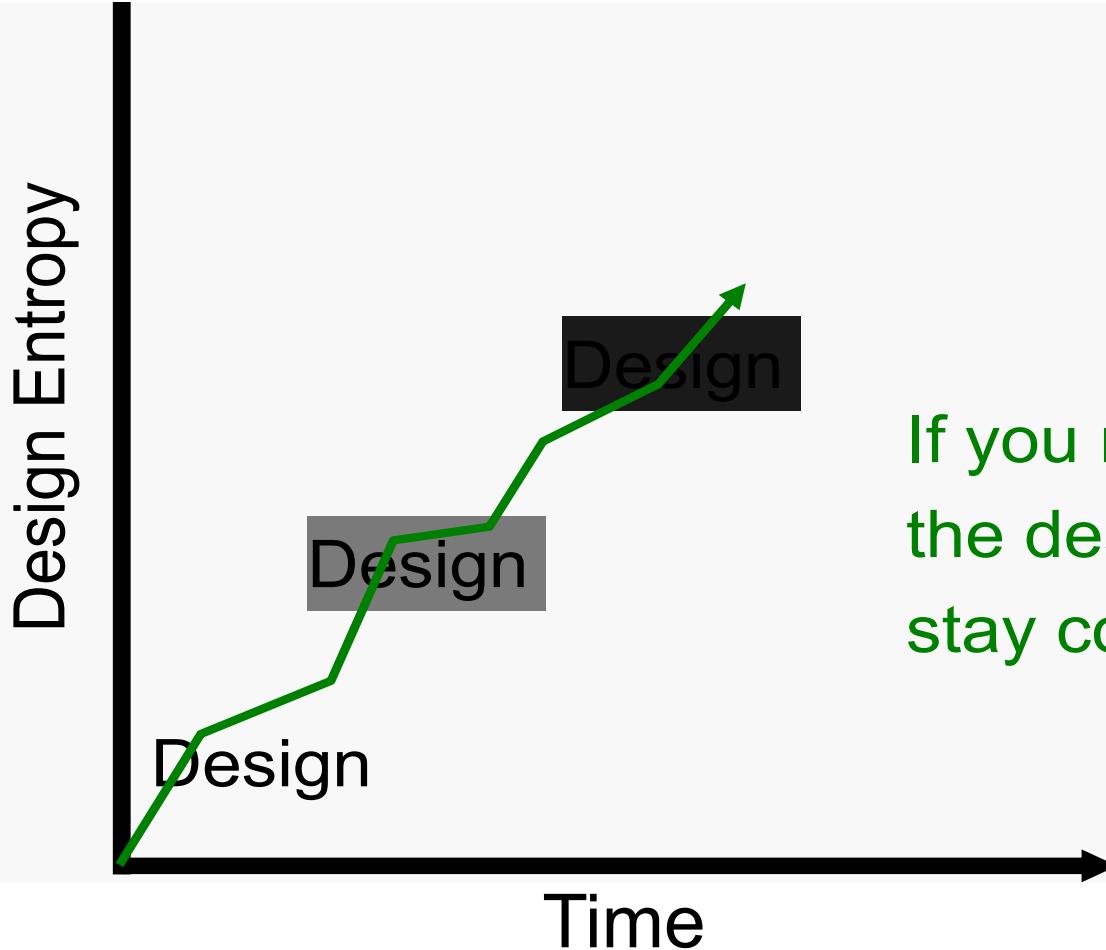
Source: Mike Lutz, RIT

... To this



Source: Mike Lutz, RIT

Design Entropy Vs Time



If you no longer can see
the design, how can you
stay consistent to it?

What causes Design Smells?

- Violation of design principles
 - `java.util.Calender`
 - Violates Single responsibility principle (overloaded with date functionality and time functionality)
 - `java.util.Stack`
 - extends `java.util.Vector`
 - Do Stack and Vector share a IS-A relationship?
 - Violates principle of hierarchy (Broken Hierarchy – substitutability is broken)

What causes Design Smells?

- Inappropriate use of patterns
- Language limitations
- Procedural thinking in OO
- Viscosity
 - Software viscosity
 - Environment viscosity
- Non-adherence to best practices and processes

Design Smell classification

Abstraction

- Missing Abstraction
- Imperative Abstraction
- Incomplete Abstraction
- Multifaceted Abstraction
- Unnecessary Abstraction
- Unutilized Abstraction
- Duplicate Abstraction

Modularization

- Broken Modularization
- Insufficient Modularization
- Cyclically-dependent Modularization
- Hub-like Modularization

Design Smell classification

Encapsulation

- Deficient Encapsulation
- Leaky Encapsulation
- Missing Encapsulation
- Unexploited Encapsulation

Modularization

- Missing Hierarchy
- Unnecessary Hierarchy
- Unfactored Hierarchy
- Wide Hierarchy
- Speculative Hierarchy
- Deep Hierarchy
- Rebellious Hierarchy
- Broken Hierarchy
- Multipath Hierarchy
- Cyclic Hierarchy

Code Smells Within Classes

■ Comments

- Are the comments necessary?
- Do they explain "why" and not "what"?
- Can you refactor the code so the comments aren't required?
- Remember, you're writing comments for people, not machines.

■ Long Method

- Shorter method is easier to read, easier to understand, and easier to troubleshoot.
- Refactor long methods into smaller methods if you can

■ Long Parameter List

- The more parameters a method has, the more complex it is.
- Limit the number of parameters you need in a given method, or use an object to combine the parameters.

■ Duplicated code

- Stamp out duplication whenever possible.
- [Don't Repeat Yourself!](#)

Code Smells Within Classes

■ Conditional Complexity

- large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time.
- Consider alternative object-oriented approaches such as decorator, strategy, or state.

■ Combinatorial Explosion

- Lots of code that does *almost* the same thing.. but with tiny variations in data or behavior.
- This can be difficult to refactor-- perhaps using generics or an interpreter?

■ Large Class

- Large classes, like long methods, are difficult to read, understand, and troubleshoot.
- Large class can be restructured or broken into smaller

Code Smells Within Classes

■ Uncommunicative Name

- ❑ Does the name of the method succinctly describe what that method does? Could you read the method's name to another developer and have them explain to you what it does?

■ Inconsistent Names

- ❑ set of standard terminology and stick to it throughout your methods.

■ Dead Code

- ❑ Ruthlessly delete code that isn't being used

■ Speculative Generality

- ❑ Write code to solve today's problems, and worry about tomorrow's problems when they actually materialize.
- ❑ Everyone loses in the "what if.." school of design.

Code Smells Between Classes

■ Alternative Classes with Different Interfaces

- ❑ If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface.

■ Primitive Obsession

- ❑ If data type is sufficiently complex, write a class to represent it.

■ Data Class

- ❑ Avoid classes that passively store data.
- ❑ Classes should contain data *and* methods to operate on that data, too.

■ Data Clumps

- ❑ If you always see the same data hanging around together, maybe it belongs together.
- ❑ Consider rolling the related data up into a larger class.

Code Smells Between Classes

- **Refused Bequest**
 - Inherit from a class but never use any of the inherited functionality
- **Inappropriate Intimacy**
 - Classes that spend too much time together, or classes that interface in inappropriate ways.
 - Classes should know as little as possible about each other
- **Indecent Exposure**
 - Classes that unnecessarily expose their internals.
 - Aggressively refactor classes to minimize their public surface.
 - You should have a compelling reason for every item you make public. If you don't, hide it.
- **Feature Envy**
 - Methods that make extensive use of another class may belong in another class.
 - Move the method to the class it is so envious

Code Smells between Classes

- **Lazy Class**
 - Classes should pull their weight.
 - If a class isn't doing enough to pay for itself, it should be collapsed or combined into another class.
- **Message Chains**
 - Long sequences of method calls or temporary variables to get routine data.
 - Intermediaries are dependencies in disguise.
- **Middle Man**
 - If a class is delegating all its work., then cut out the middleman.
 - Beware classes that are merely wrappers over other classes or existing functionality in the framework.
- **Divergent Change**
 - If changes to a class that touch completely different parts of the class, it may contain too much unrelated functionality.
 - Isolate the parts that changed in another class.

Code Smells between Classes

- Shotgun Surgery
 - If a change in one class requires cascading changes in several related classes
- Parallel Inheritance Hierarchies
 - Every time you make a subclass of one class, you must also make a subclass of another.
 - Consider folding the hierarchy into a single class.
- Solution Sprawl
 - If it takes five classes to do anything useful, you might have solution sprawl.
 - Consider simplifying and consolidating your design.

Refactoring

- As a software system grows, the overall design often suffers
- In the short term, working in the existing design is cheaper than doing a redesign
- In the long term, the redesign decreases total costs
 - Extensions
 - Maintenance
 - Understanding
- Refactoring is a set of techniques that reduce the short-term pain of redesigning
 - Not adding functionality
 - Changing structure to make it easier to understand and extend

The Scope of Refactoring

- Small steps:
 - Rename a method
 - Move a field from one class to another
 - Merge two similar methods in different classes into one common method in a base class
- Each individual step is small, and easily verified/tested
- The composite effect can be a complete transformation of a system

Principles

- Don't refactor and extend a system at the same time
 - Make a clear separation between the two activities
- Have good tests in place before you begin refactoring
 - Run the tests often
 - Catch defects immediately
- Take small steps
 - Many localized changes result in a larger-scale change
 - Test after each small step

When Should You Refactor?

- You're extending a system, and realize it could be done better by changing the original structure
 - Stop and refactor first
- The code is hard to understand
 - Refactor to gain understanding, and leave the code better than it was

Refactoring and OOD

- The refactoring literature is written from a coding perspective
- Many of the operations still apply at design time
- It helps if you have an appropriate level of detail in the design
 - Too much, and you may as well code
 - Too little, and you can't tell what's happening



AntiPatterns



AntiPatterns

- A pattern of practice that is commonly found in use
- A pattern which when practiced usually results in *negative* consequences
- Patterns defined in several categories of software development
 - Design
 - Architecture
 - Project Management

Purpose for AntiPatterns

- Identify problems
- Develop and implement strategies to fix
 - Work incrementally
 - Many alternatives to consider
 - Beware of the cure being worse than the disease

Pattern Vs AntiPattern

■ Patterns

- Usually bottom up
- Begin with recurring solution
- Then the forces and context
- Usually leads to one solution

■ AntiPatterns

- Top down
- Begin with commonly recurring practice
- Obvious negative consequences
- Symptoms are past and present; consequences go into the future

Software Design AntiPatterns

■ AntiPatterns

- The Blob
- Lava Flow
- Functional Decomposition
- Poltergeists
- Golden Hammer
- Spaghetti Code
- Cut-and-Paste Programming

■ Mini-AntiPatterns

- Continuous Obsolescence
- Ambiguous Viewpoint
- Boat Anchor
- Dead End
- Input Kludge
- Walking through a Minefield
- Mushroom Management

The Blob

- AKA
 - Winnebago, The God Class, Kitchen Sink Class
- Causes
 - Sloth, haste
- Unbalanced Forces:
 - Management of Functionality, Performance, Complexity
- Anecdotal Evidence:
 - “This is the class that is really the *heart* of our architecture.”

The Blob (2)

- Like the blob in the movie can consume entire object-oriented architectures
- Symptoms
 - Single controller class, multiple simple data classes
 - No object-oriented design, i.e. all in main
 - Start with a legacy design
- Problems
 - Too complex to test or reuse
 - Expensive to load into system

Causes

- Lack of OO architecture
- Lack of any architecture
- Lack of architecture enforcement
- Limited refactoring intervention
- Iterative development
 - Proof-of-concept to prototype to production
 - Allocation of responsibilities not repartitioned

Solution

- Identify or categorize related attributes and operations
- Migrate functionality to data classes
- Remove far couplings and migrate to data classes

Lava Flow

- AKA
 - Dead Code
- Causes
 - Avarice, Greed, Sloth
- Unbalanced Forces
 - Management of Functionality, Performance, Complexity

Symptoms and Consequences

- Unjustifiable variables and code fragments
- Undocumented complex, important-looking functions, classes
- Large commented-out code with no explanations
- Lots of “to be replaced” code
- Obsolete interfaces in header files
- Proliferates as code is reused

Causes

- Research code moved into production
- Uncontrolled distribution of unfinished code
- No configuration management in place
- Repetitive development cycle

Solution

- Don't get to that point
- Have stable, well-defined interfaces
- Slowly remove dead code; gain a full understanding of any bugs introduced
- Strong architecture moving forward

Functional Decomposition

- AKA
 - No OO
- Root Causes
 - Avarice, Greed, Sloth
- Unbalanced Forces
 - Management of Complexity, Change
- Anecdotal Evidence
 - “This is our ‘main’ routine, here in the class called Listener.”

Symptoms and Consequences

- Non-OO programmers make each subroutine a class
- Classes with functional names
 - Calculate_Interest
 - Display_Table
- Classes with single method
- No leveraging of OO principles
- No hope of reuse

Causes

- Lack of OO understanding
- Lack of architecture enforcement
- Specified disaster

Solution

- Perform analysis
- Develop design model that incorporates as much of the system as possible
- For classes outside model:
 - Single method: find home in existing class
 - Combine classes

Poltergeists

- AKA
 - Gypsy, Proliferation of Classes
- Root Causes
 - Sloth, Ignorance
- Unbalanced Forces
 - Management of Functionality, Complexity
- Anecdotal Evidence
 - “I’m not exactly sure what this class does, but it sure is important.”

Symptoms and Consequences

- Transient associations that go “bump-in-the-night”
- Stateless classes
- Short-lived classes that begin operations
- Classes with control-like names or suffixed with *manager* or *controller*. Only invoke methods in other classes.

Causes

- Lack of OO experience
- Maybe OO is incorrect tool for the job. “There is no right way to do the wrong thing.”

Solution

- Remove Poltergeist altogether
- Move controlling actions to related classes

Cut-and-Paste Programming

- AKA
 - Clipboard Coding
- Root Causes
 - Sloth
- Unbalanced Forces
 - Management of Resources, Technology Transfer
- Anecdotal Evidence
 - “Hey, I thought you fixed that bug already, so why is it doing this again?” “Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!”

Symptoms and Consequences

- Same software bug reoccurs
- Code can be reused with a minimum of effort
- Causes excessive maintenance costs
- Multiple unique bug fixes develop
- Inflates LOC without reducing maintenance costs

Causes

- Requires effort to create reusable code; must reward for long-term investment
- Context or intent of module not preserved
- Development speed overshadows all other factors
- “Not-invented-here” reduces reuse
- People unfamiliar with new technology or tools just modify a working example

Solution

- Code mining to find duplicate sections of code
- Refactoring to develop standard version
- Configuration management to assist in prevention of future occurrence

Golden Hammer

- AKA
 - Old Yeller
- Root Causes
 - Ignorance, Pride, Narrow-Mindedness
- Unbalanced Forces
 - Management of Technology Transfer
- Anecdotal Evidence
 - “Our database is our architecture” “Maybe we shouldn’t have used Excel macros for this job after all.”

Symptoms and Consequences

- Identical tools for conceptually diverse problems.
“When your only tool is a hammer everything looks like a nail.”
- Solutions have inferior performance, scalability and other ‘ilities’ compared to other solutions in the industry.
- Architecture is described by the tool set.
- Requirements tailored to what tool set does well.

Causes

- Development team is highly proficient with one toolset.
- Several successes with tool set.
- Large investment in tool set.
- Development team is out of touch with industry.

Solution

- Organization must commit to exploration of new technologies
- Commitment to professional development of staff
- Defined software boundaries to ease replacement of subsystems
- Staff hired with different backgrounds and from different areas
- Use open systems and architectures



OO Design patterns

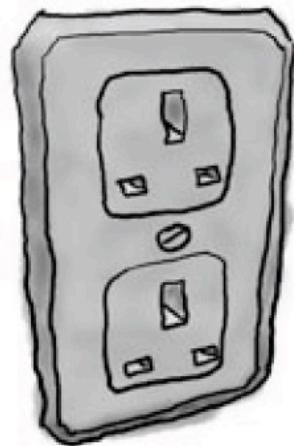
Acknowledgement: Freeman & Freeman



Adapter pattern

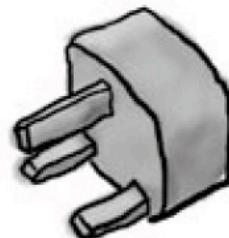
Example Scenario

European Wall Outlet



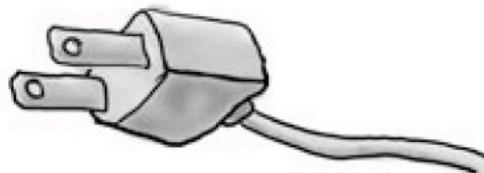
The European wall outlet exposes one interface for getting power

AC Power Adapter



The adapter converts one interface into another

Standard AC Plug



The US laptop expects another interface

Adapter – in action

```
public interface Duck {  
    public void quack();  
    public void fly();  
}
```

Here is a subclass of Duck

```
public class MallardDuck implements Duck {  
    public void quack() {  
        System.out.println("Quack");  
    }  
    public void fly() {  
        System.out.println("Flying");  
    }  
}
```

Adapter – in action

```
public interface Turkey {  
    public void gobble();  
    public void fly();  
}
```

Here is a subclass of Turkey

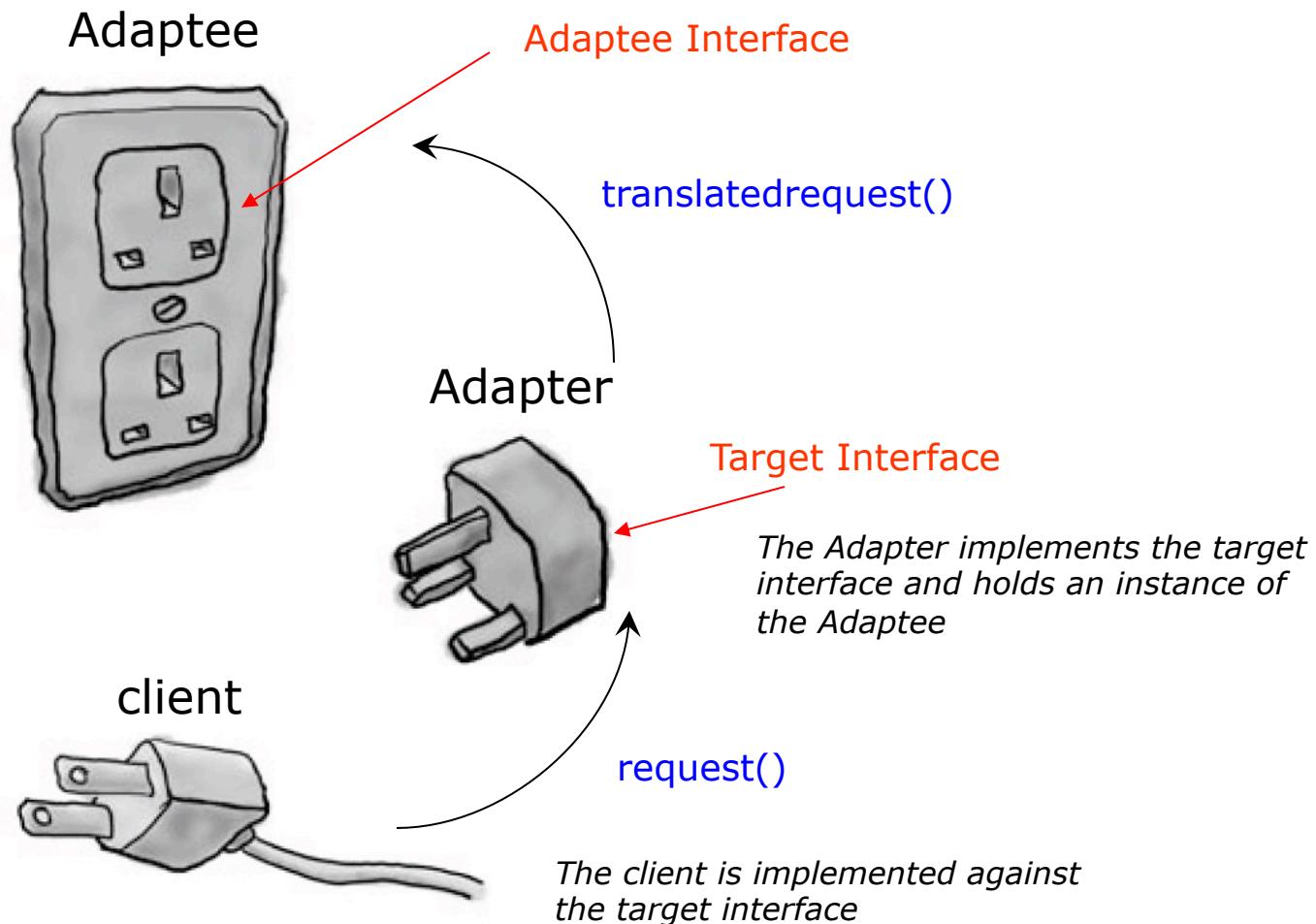
```
public class WildTurkey implements Turkey{  
    public void gobble() {  
        System.out.println("Gobble gobble");  
    }  
    public void fly() {  
        System.out.println("Flying short distance");  
    }  
}
```

Adapter – in action

- If Turkey objects are to be used instead of Duck objects

```
public class TurkeyAdapter implements Duck{  
    Turkey turkey;  
  
    public TurkeyAdapter(Turkey turkey) {  
        this.turkey = turkey;  
    }  
  
    public void quack() {  
        turkey.gobble();  
    }  
    public void fly() {  
        for (int i=0; i<10; i++) {  
            turkey.fly();  
        }  
    }  
}
```

Adapter Pattern Explained



Discussion Questions

- How much “adapting” does an adapter need to do? It seems like if I need to implement a large target interface, I could have a **LOT** of work on my hands
- Does an adapter always wrap one and only one class?
- What if I have old and new parts of my system, the old parts expect the old vendor interface, but we’ve already written the new parts to use the new vendor interface? It is going to get confusing using an adapter here and the unwrapped interface there. Wouldn’t I be better off just writing my older code and forgetting the adapter?

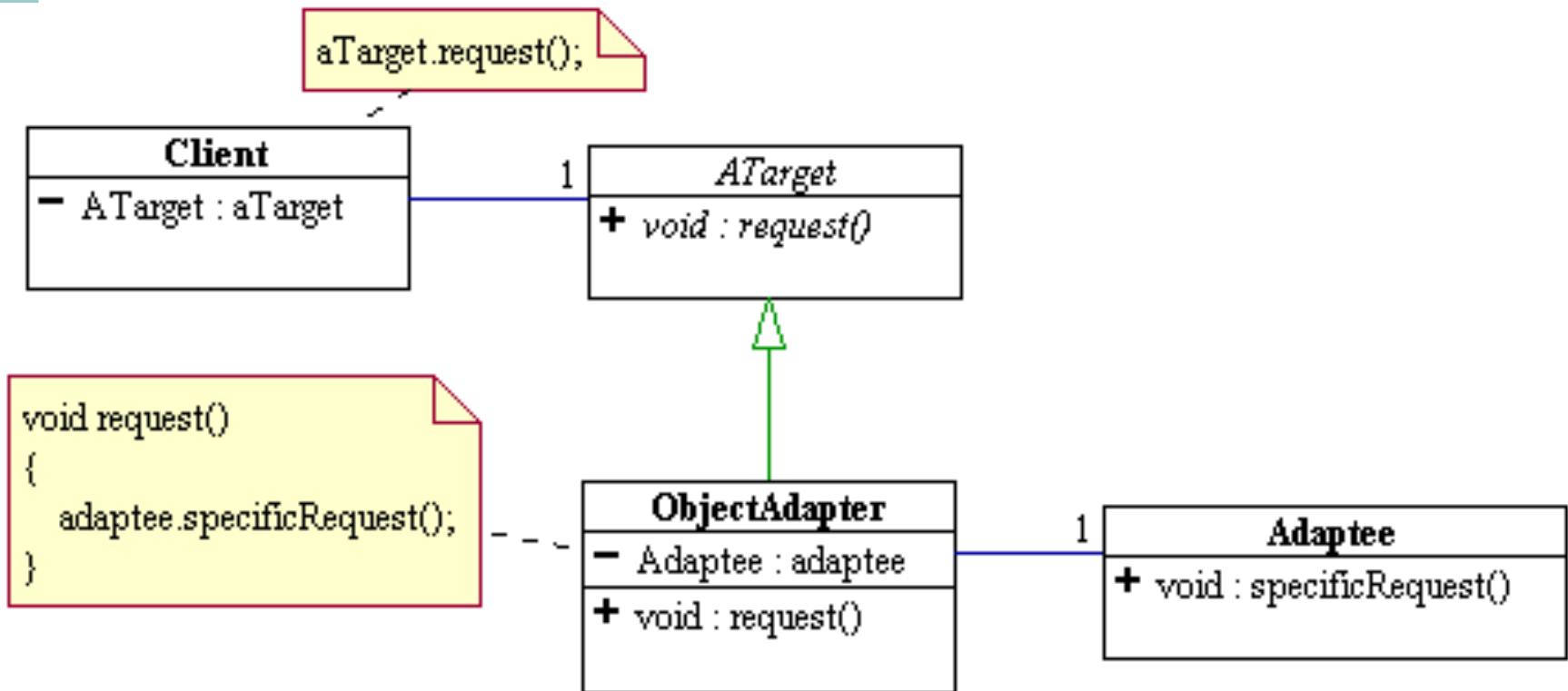
Adapter Pattern

- An adapter pattern converts the interface of a class into an interface that a client expects
- Adapters allow incompatible classes to work together
- Adapters can extend the functionality of the adapted class
- Commonly called “glue” or “wrapper”

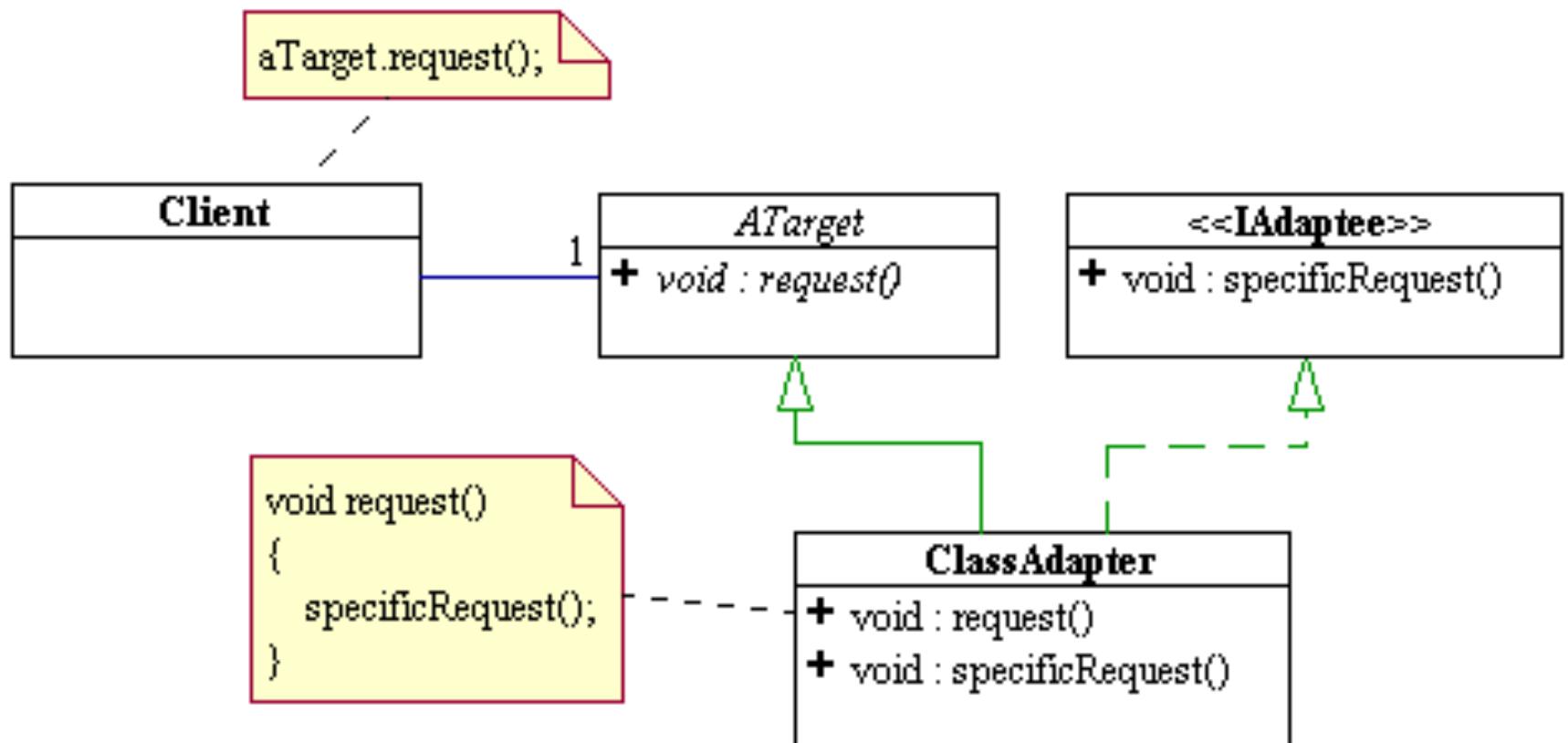
When to Use

- Need to adapt the interface of an existing class to satisfy client interface requirements
 - Adapting Legacy Software
 - Adapting 3rd Party Software

Object Adapter Pattern

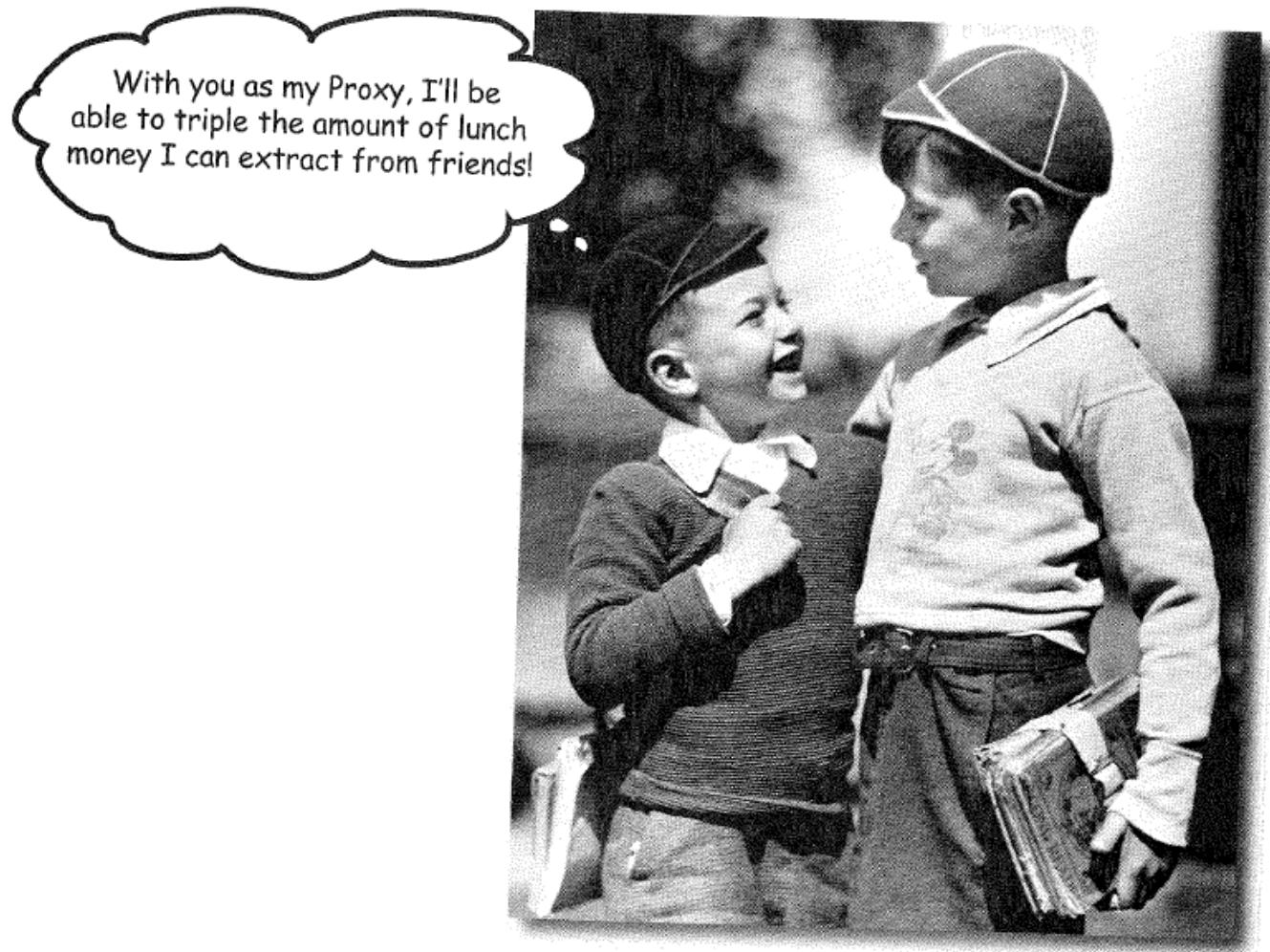


Class Adapter Pattern



Proxy pattern

I like proxies...



The Problems

- Expensive & inexpensive pieces of state
 - Example: Large image
 - Inexpensive: size & location of drawing
 - Expensive: load & display
- Remote objects (e.g., another system)
 - Want to access it as if it were local
 - Want to hide all the required communications
 - Example: Java RMI
- Object with varying access rights
 - Some clients can access anything
 - Other clients have subset of functionality available

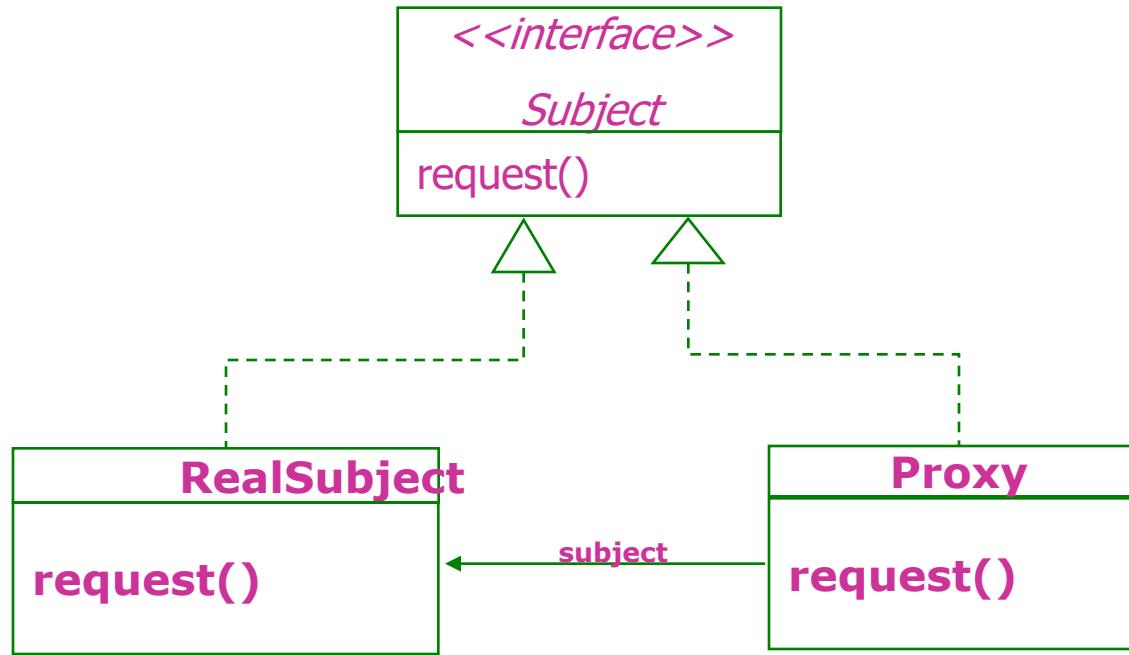
The Design Goal

- In all these cases desire access to object as if it is directly available
- For efficiency, simplicity, or security, put a *proxy* object in front of the real object
- We have a stand-in for the real object to control how the real object behaves

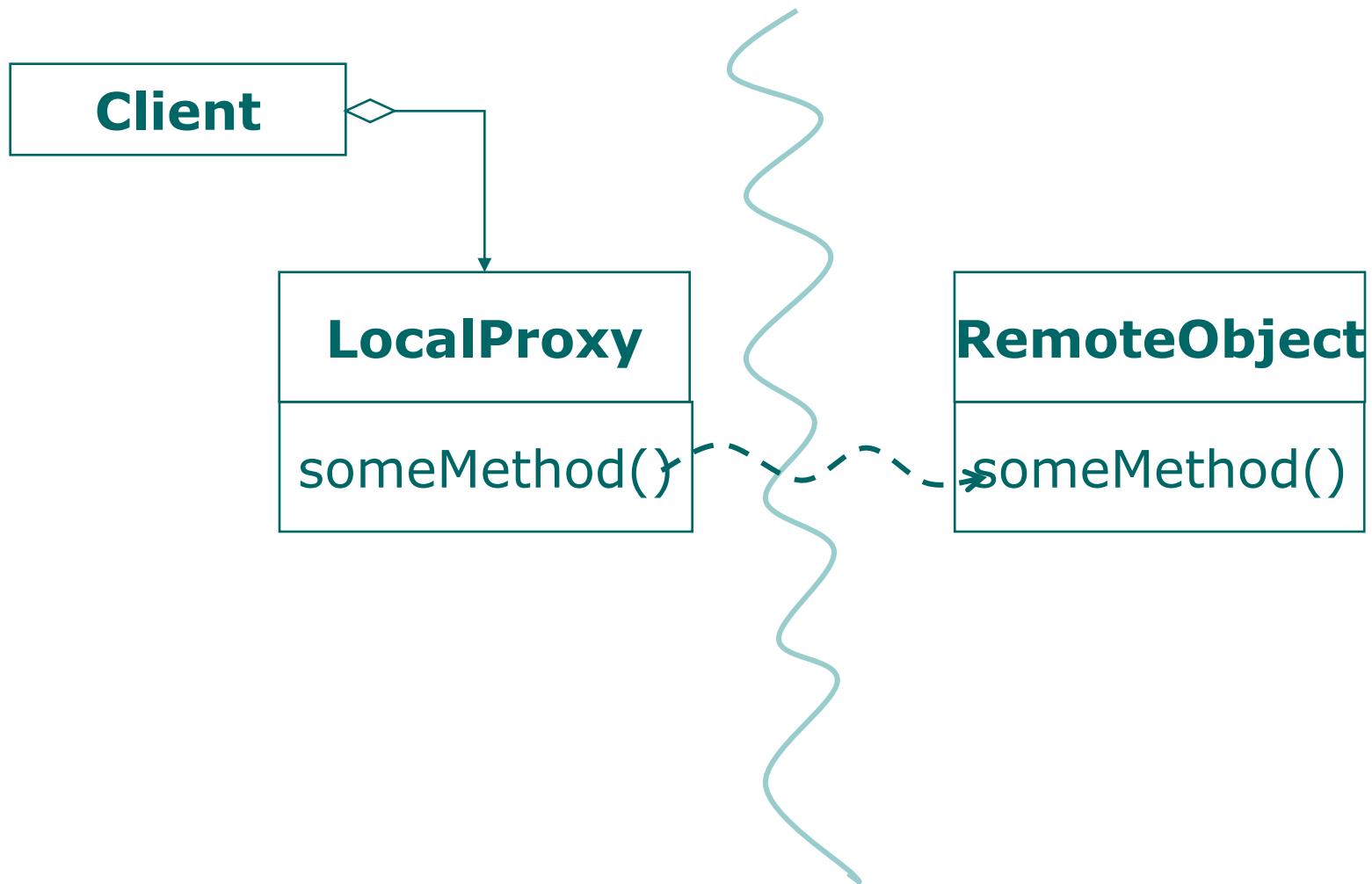
Proxy pattern Defined

- Proxy patterns provides a surrogate or placeholder for another object to control access to it
 - **Remote proxy** controls access to a remote object
 - **Virtual proxy** controls access to a resource that is expensive to create
 - **Protection proxy** controls access to a resource based on access rights

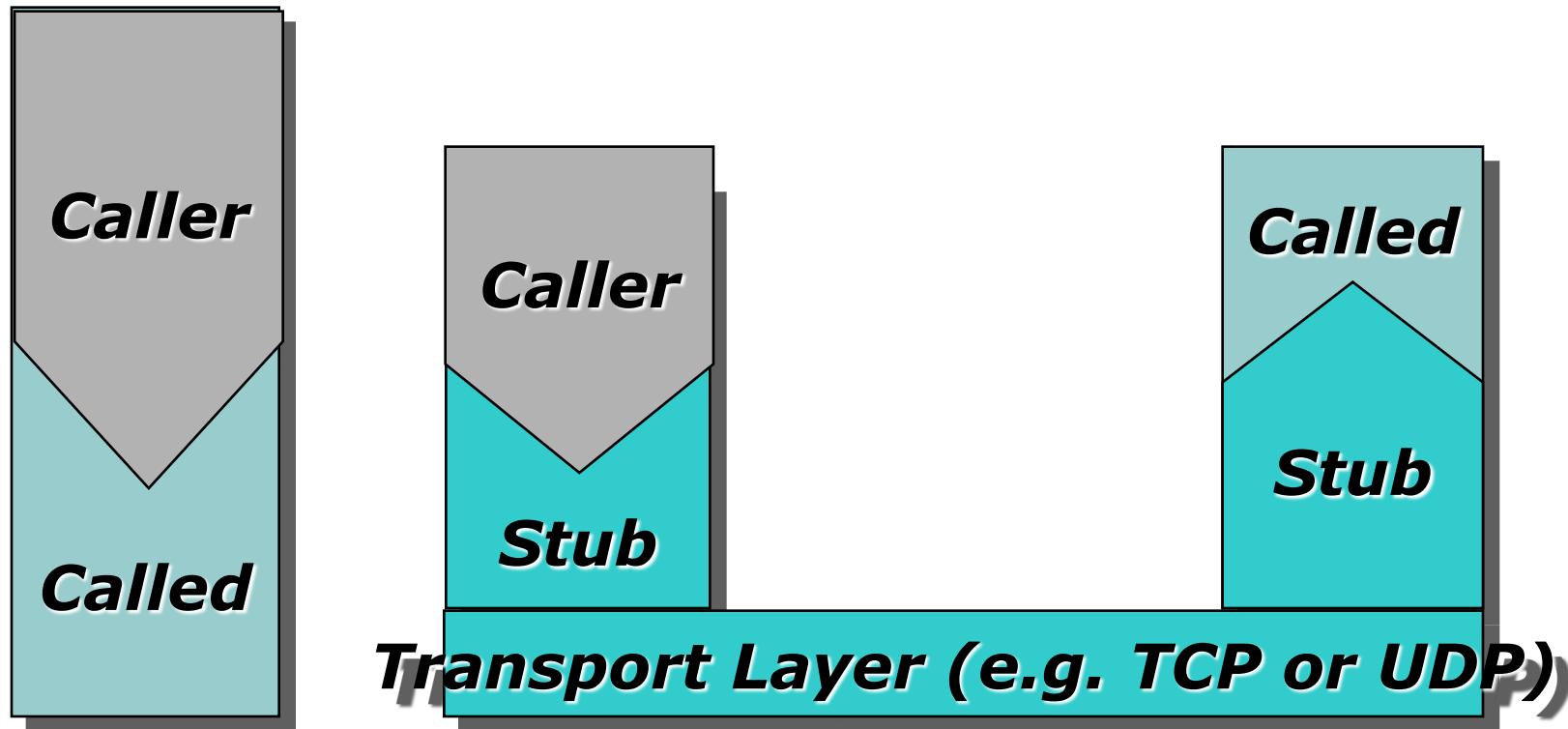
Proxy pattern structure



Example: Accessing Remote Object



Java RMI, the big picture



Categories of Proxies

- **Remote proxy** - as above
 - Local representative for something in a different address space
 - Java RMI tools help set these up automatically
 - Object brokers handle remote objects (CORBA or DCOM)
- **Virtual proxy**
 - Stand-in for an object that is expensive to implement or completely access
 - Example – image over the net
 - May be able to access some state (e.g., geometry) at low cost
 - Defer other high costs until it must be incurred
- **Protection proxy**
 - Control access to the "real" object
 - Different proxies provide different rights to different clients
 - For simple tasks, can do via multiple interfaces available to clients
 - For more dynamic checking, need a front-end such as a proxy

Variants of Proxies

- **Firewall proxy** – controls access to a set of network resources protecting the subject from “bad” clients
- **Smart Reference Proxy** – provides additional actions whenever a subject is referenced, such as counting the number of references to an object
- **Caching Proxy** – provides temporary storage for results of operations that are expensive. It can allow multiple clients to share the results to reduce computation or network latency
- **Synchronization Proxy** – provides safe access to a subject from multiple threads
- **Complexity hiding Proxy** – hides the complexity of and controls access to a complex set of classes. Sometimes called as **Façade proxy** for obvious reasons
- **Copy-on-Write Proxy** – controls the copying of an object by deferring the copying of an object until it is required by a client. This is a variant of the **Virtual proxy**.

Consequences

- Advantages traded-off against cost of extra level of indirection
- Can hide information about the real object
 - Where it is (remote proxy)
 - When it is loaded (virtual proxy)
 - How it is accessed (protection proxy)
- The proxy interface is either:
 - The same as the real object
 - A subset of the real object

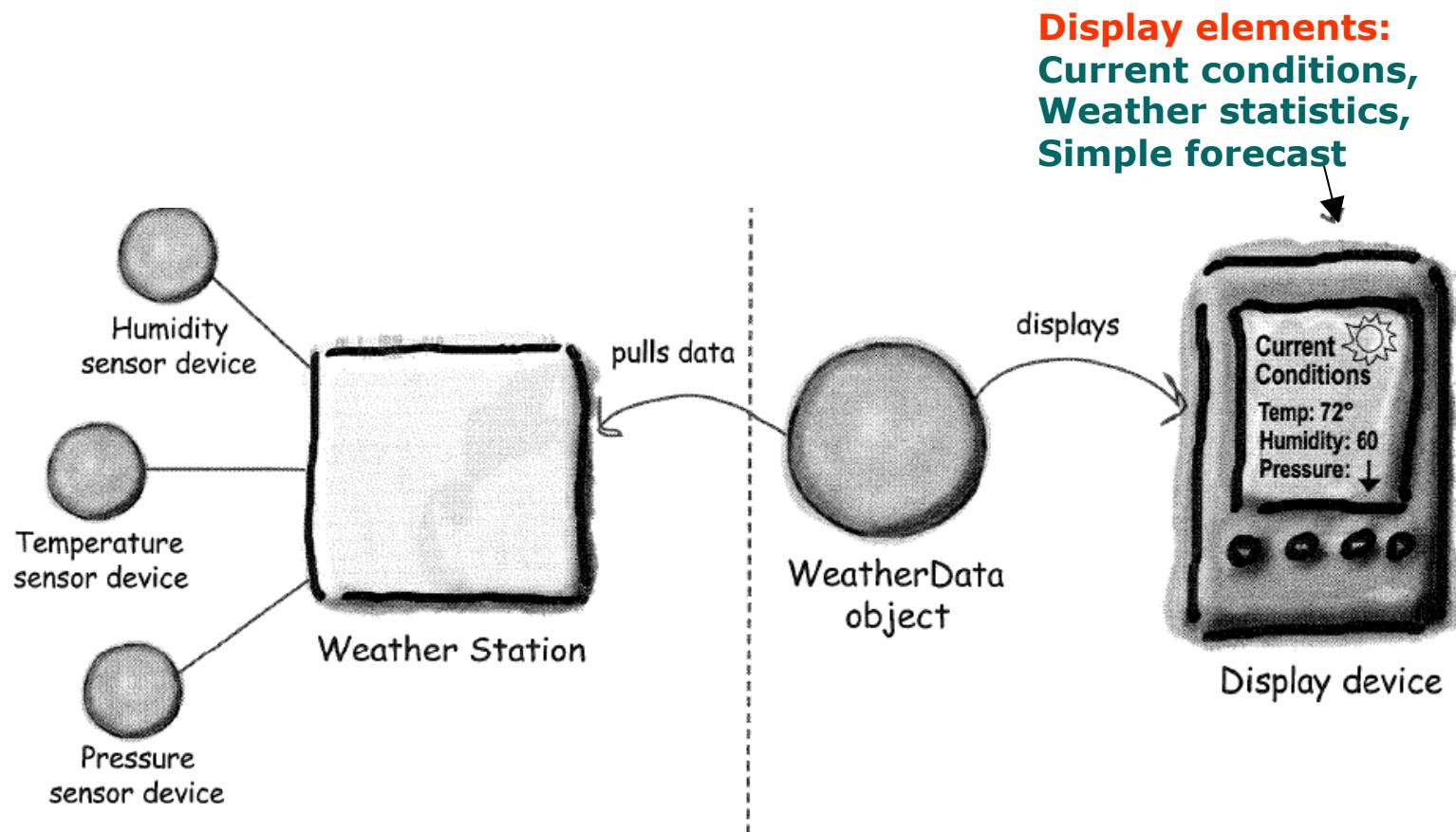
Proxies/Adapters/Facades

- Proxies and Adapters both place a stand-in object between the client and the real object
- Adapters do so to change the real object's interface
- Proxies do so to optimize access to the object via the same interface.
- Facades ease the use of sub-systems of objects



Observer pattern

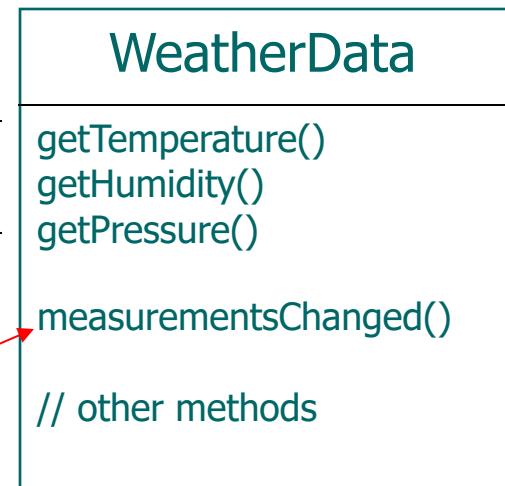
Weather Monitoring application



WeatherData class

return most recent weather measurements

Gets called whenever the measurements have been updated



WeatherData Implementation

```
public class WeatherData {  
    // instance variable declarations  
  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update (temp, humidity, pressure);  
        statisticsDisplay.update (temp, humidity, pressure);  
        forecastDisplay.update (temp, humidity, pressure);  
    }  
  
    // other WeatherData methods here  
}
```

The Problem

- Given

- Clusters of related classes
- Tight connections within each cluster of classes
- Loose state dependency between clusters

- Desired

- Keep each cluster state consistent when state changes in cluster it depends on
- Provide isolation such that changed cluster has no knowledge of specifics of dependent clusters

Example: UI & Application

- Application classes represent information being manipulated.
- UI provides way to view and alter application state.
- May have several views of state (charts, graphs, numeric tables).
- Views may be added at any time.
- How to tell views when application state has changed?

Observer Pattern

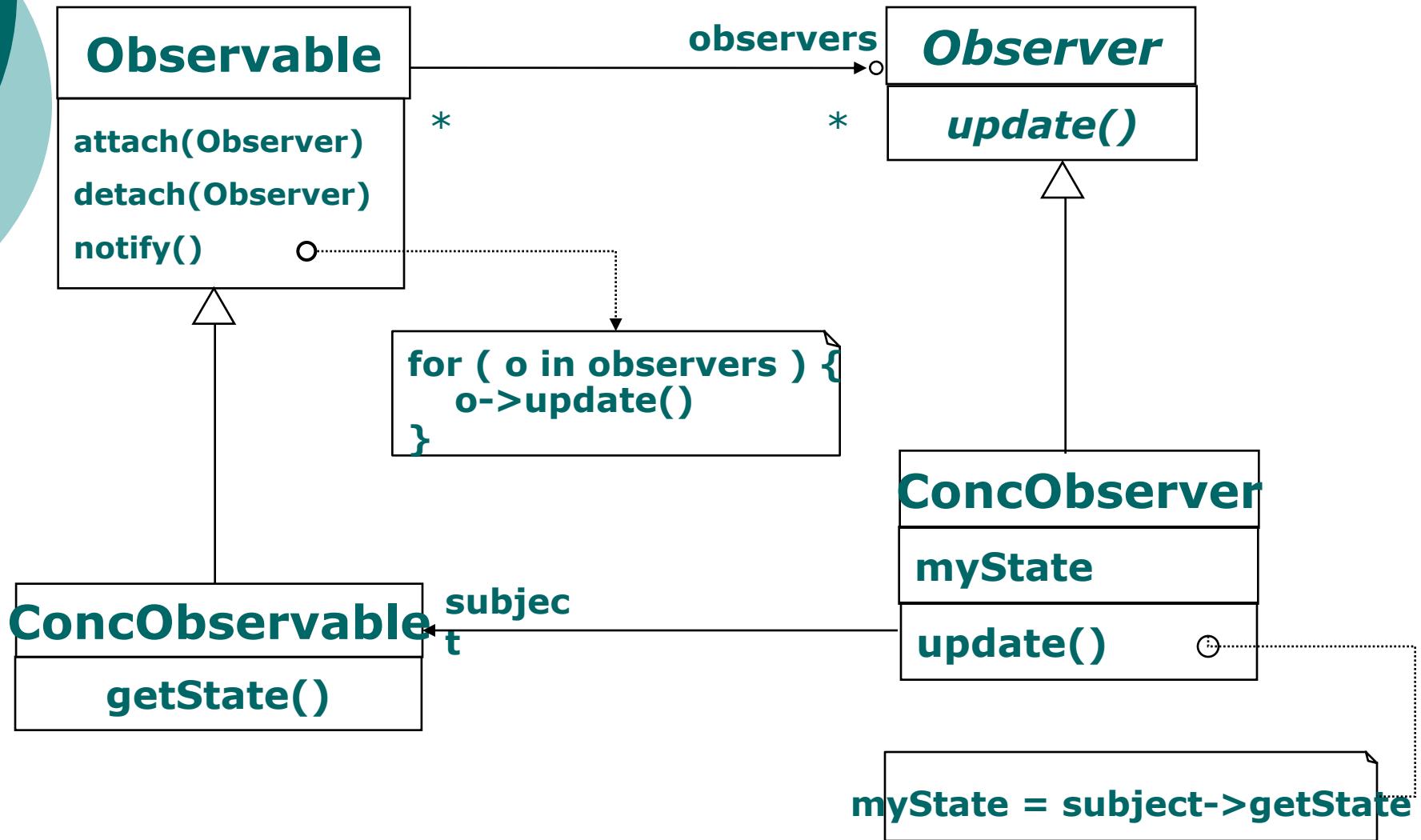
- The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically

Observer/Subject

(AKA: publish/subscribe)

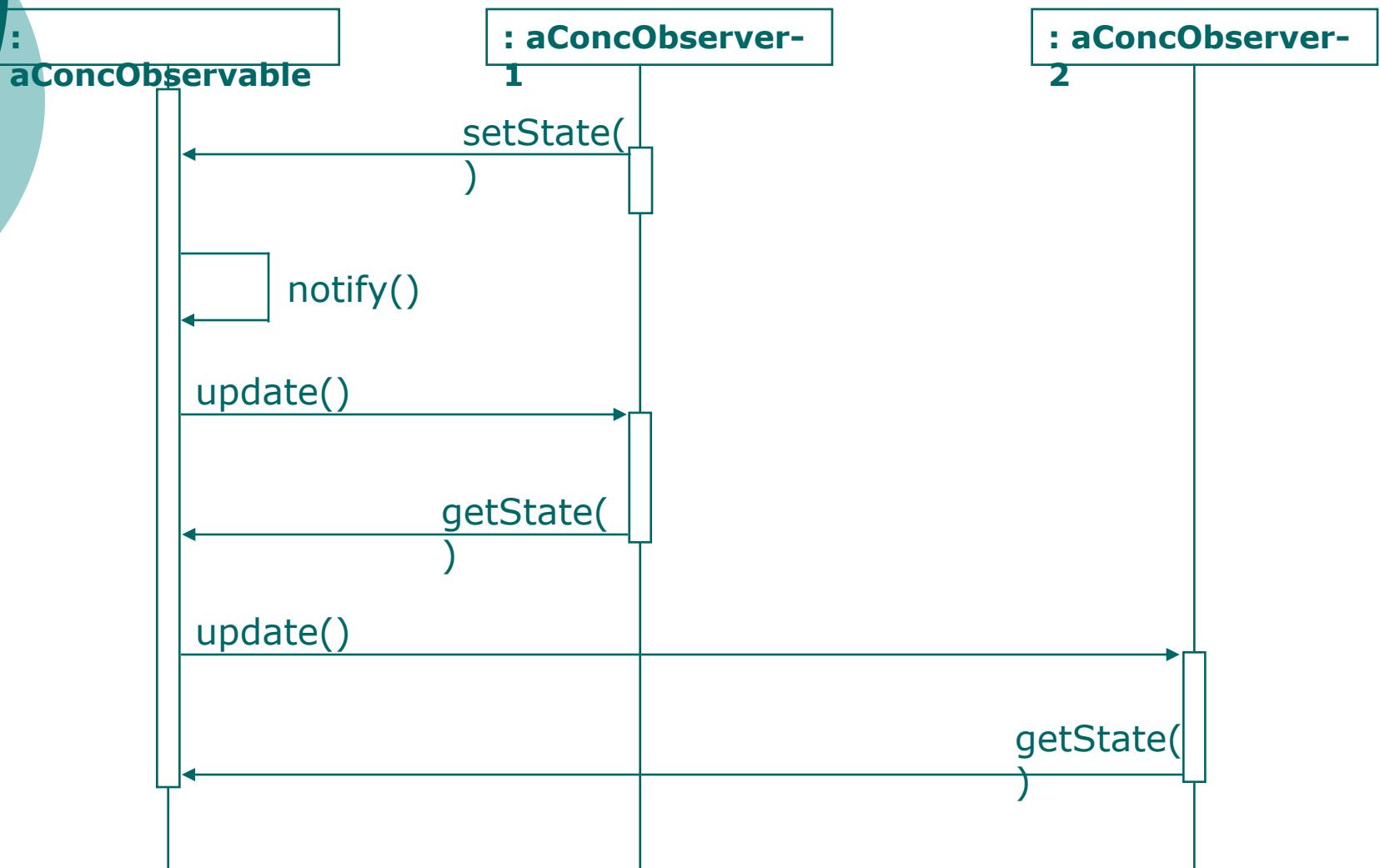
- **Observables**: objects with interesting state changes
- **Observers**: objects interested in state changes
- Observers *register* interest with observable objects.
- Observables *notify* all registered observers when state changes.

Observer Pattern - Class diagram



Weather station (example)

Interaction Diagram



When to Use Observer

- Two subsystems evolve independently but must stay in synch.
- State change in an object requires changes in an unknown number of other objects (broadcast)
- Desire loose coupling between changeable object and those interested in the change.

Consequences

- Subject/observer coupling (**loose coupling**)
 - Subject only knows it has a list of observers
 - The only thing the Subject knows about an Observer is that it implements a certain interface
 - Observers can be added at any time
 - Does not know any Observer concrete class
 - Subjects don't need to be modified to add new types of Observers
 - Subjects and Observers can be reused independently
- Supports broadcast communication
 - Observables know little about notify receivers
 - Changing observers is trivial
- Unexpected & cascading updates
 - change/notify/update -> change/notify/update
 - May be hard to tell *what* changed

Observer Pattern – Key points

- Observer pattern defines a one-to-many relationship between objects
- Subjects/Observables update observers using a common interface
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement observer interface
- You can **PUSH** or **PULL** data from the Observable when using the pattern (pull is considered more “correct”)
- Don’t depend on specific order of notification for your observers

Composite pattern

The Problem

- Problem
 - Have simple primitive component classes that collect into larger *composite* components
- Desire
 - Treat composites like primitives
 - Support composite sub-assemblies
 - Operations (usually) recurse to subassemblies
- Solution
 - Build composites from primitive elements

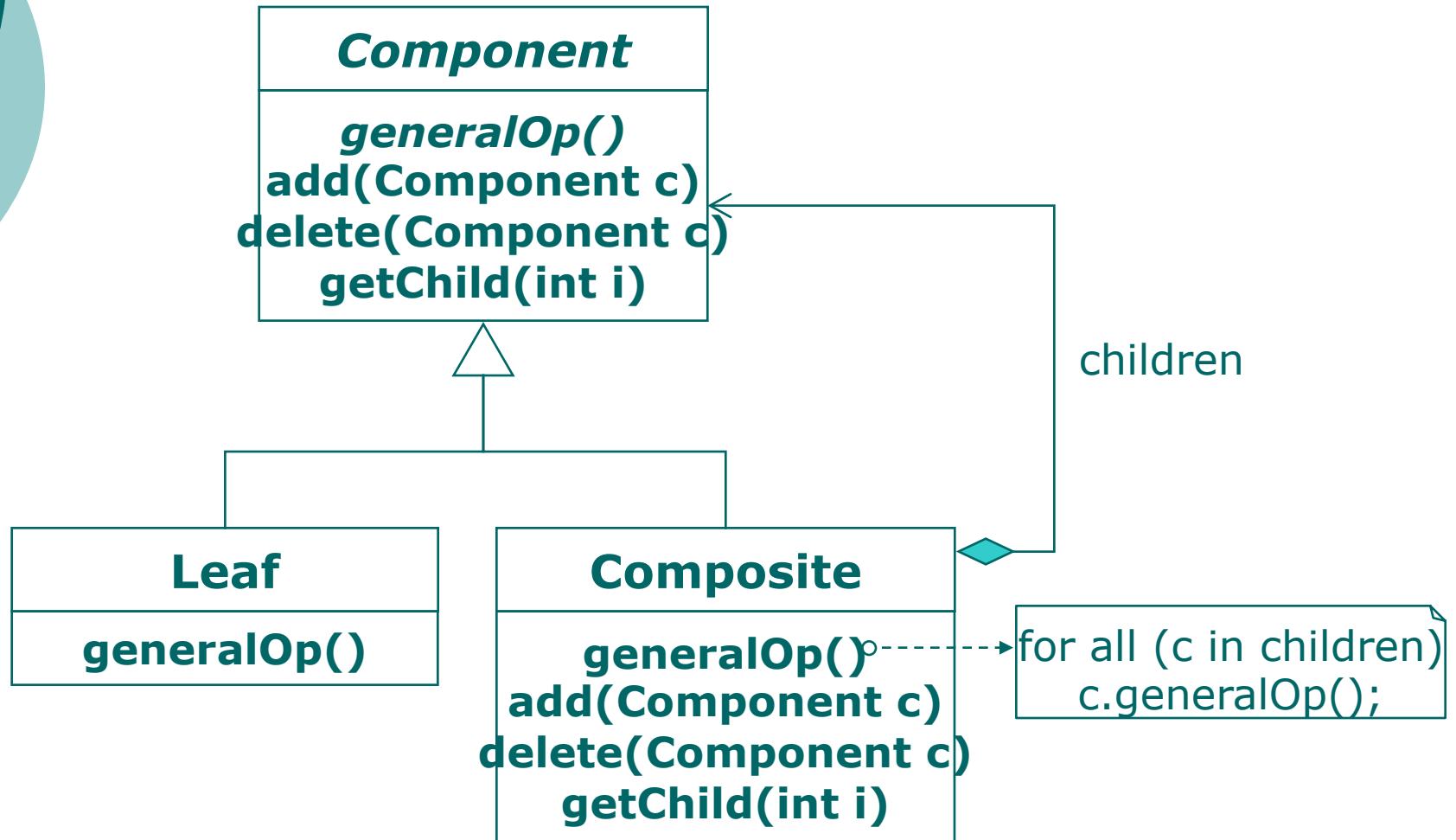
Examples - 1

- File systems
 - Primitives = text files, binary files, device files, etc.
 - Composites = directories (w/subdirectories)
- Make file dependencies
 - Primitives = leaf targets with no dependents
 - Composites = targets with dependents
- Menus
 - Primitives = menu entries
 - Composites = menus (w/submenus)

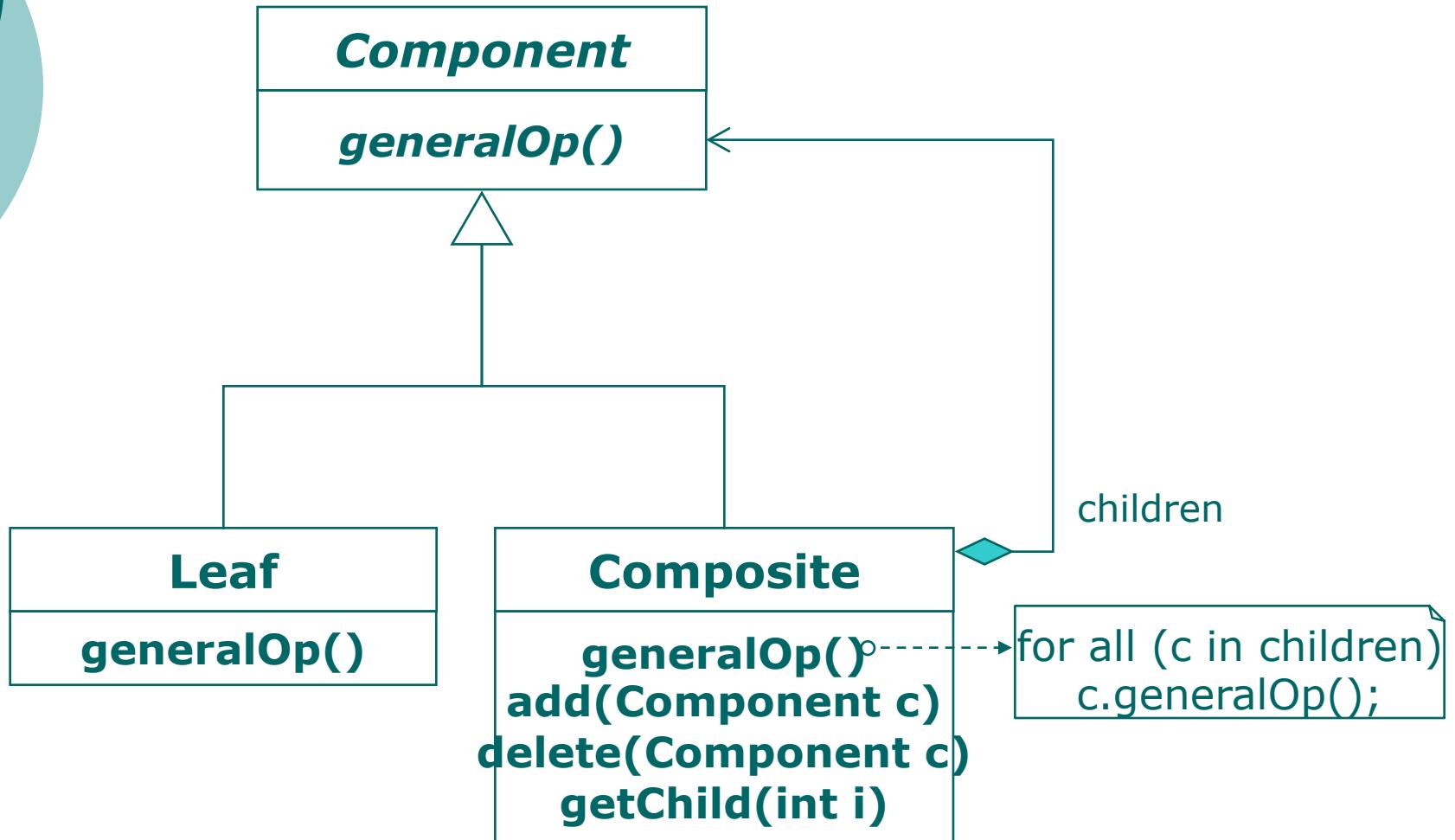
Examples - 2

- GUI Toolkits
 - Primitives = basic components (buttons, textareas, listboxes, etc).
 - Composites = frames, dialogs, panels.
- Drawing Applications
 - Primitives = lines, strings, polygons, etc.
 - Composites = groupings treated as unit.
- HTML/XML
 - Pages as composites of links (hypertext)
 - Pages as collections of paragraphs (with subparagraphs for lists, etc.)

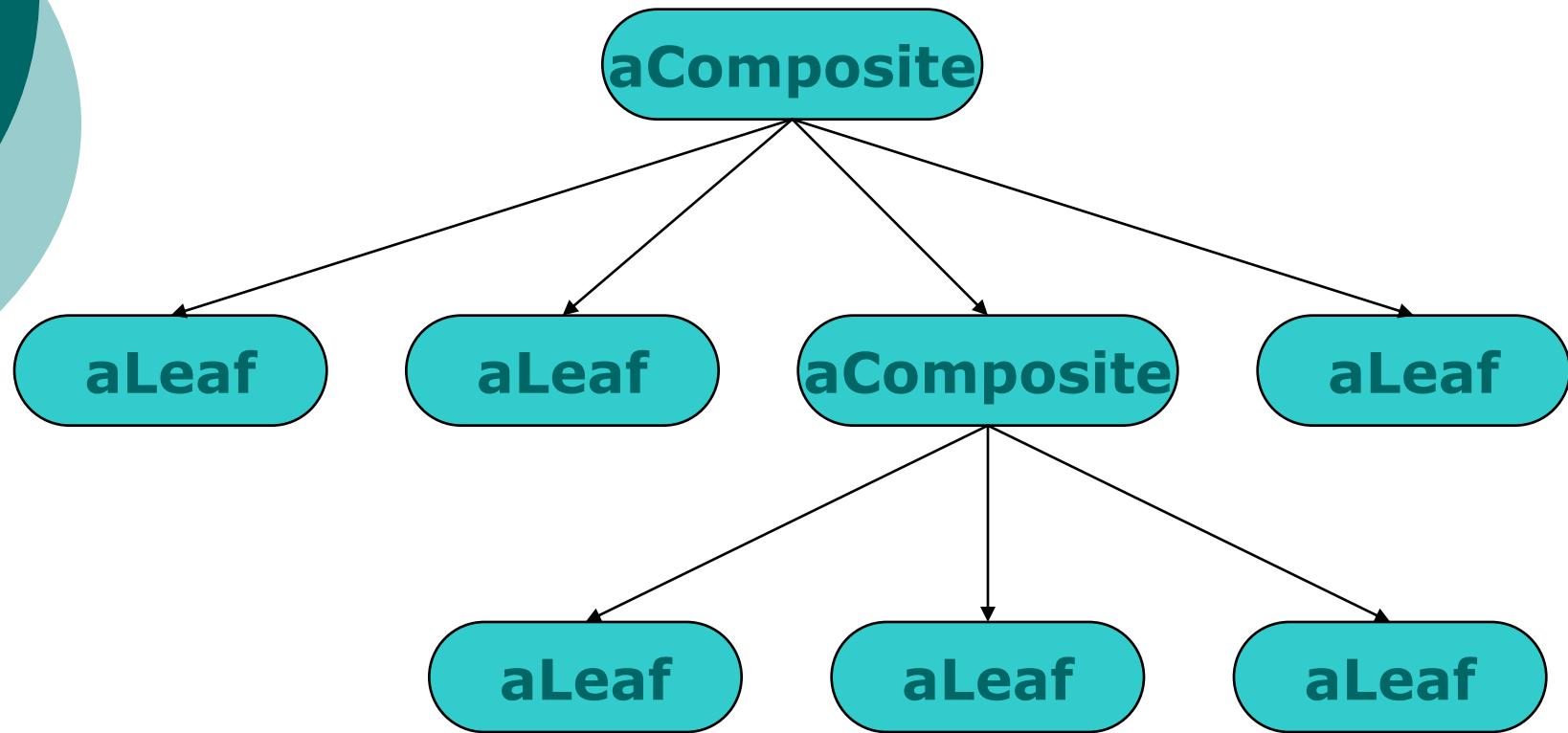
Class Diagram (Alternative 1)



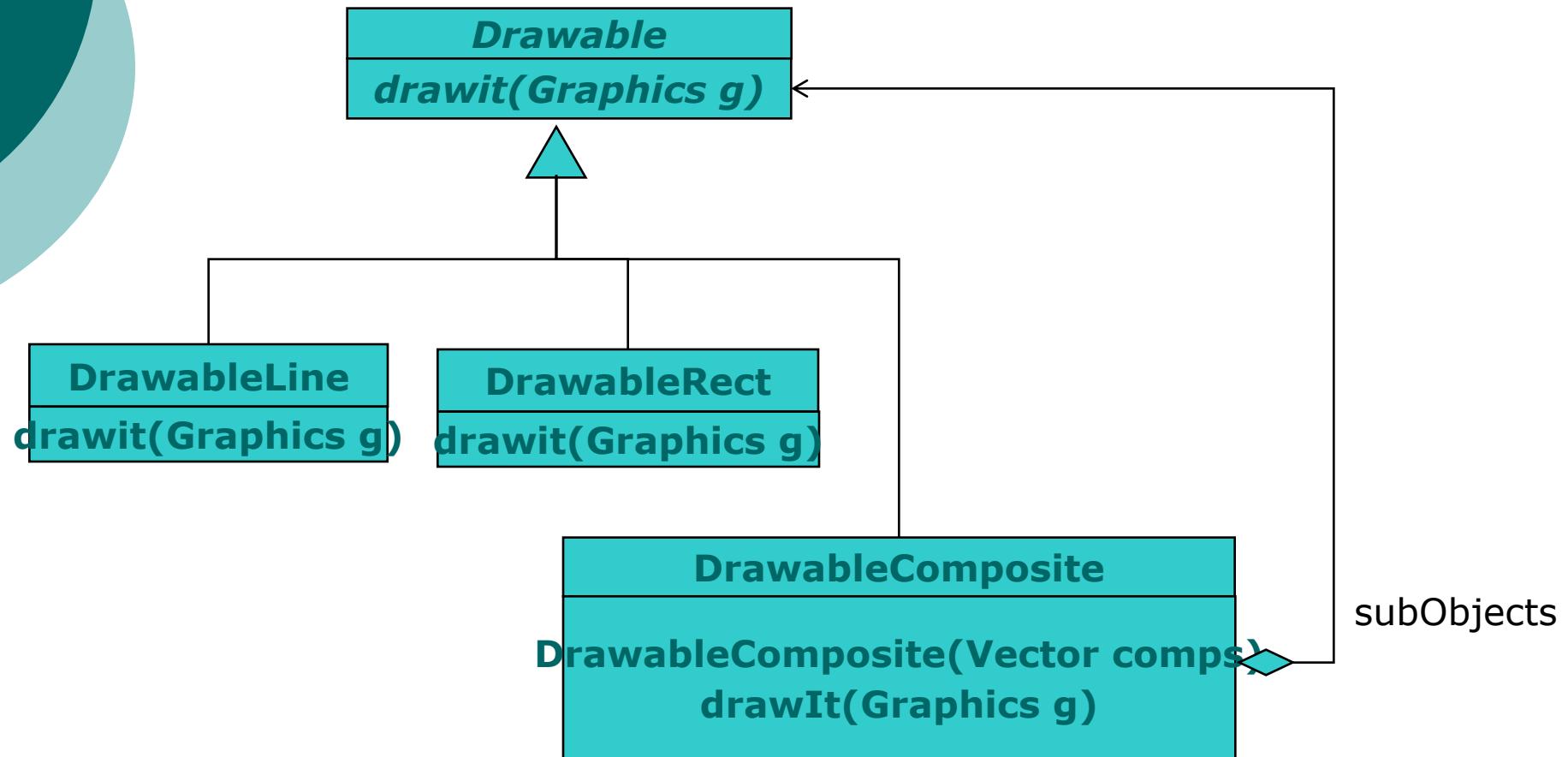
Class Diagram (Alternative 2)



Example Object Structure



Example: Drawable Figures



Discussion

- Use to model whole/part hierarchies
- Clients (usually) ignore differences between primitives & composites
- Clients access (most) components via the generic interfaces
 - Primitives implement request directly
 - Composites can handle directly or forward
- Arbitrary composition to indefinite depth
 - Tree structure – no sharing of nodes
 - General digraph – supports sharing, multiple parents – be careful!
- Eases addition of new components
 - Almost automatic
- Overly general design?



Iterator pattern

Acknowledgement: Eric Braude

Let's try this...

Data structures:

- Array
- Binary Tree
- Vector
- Linked list
- Hash table

Algorithm:

- Sort
- Find
- Merge

How many permutations to develop/maintain ?

Iterators – The Problem

○ Given

- An aggregate collection of objects

○ Desired

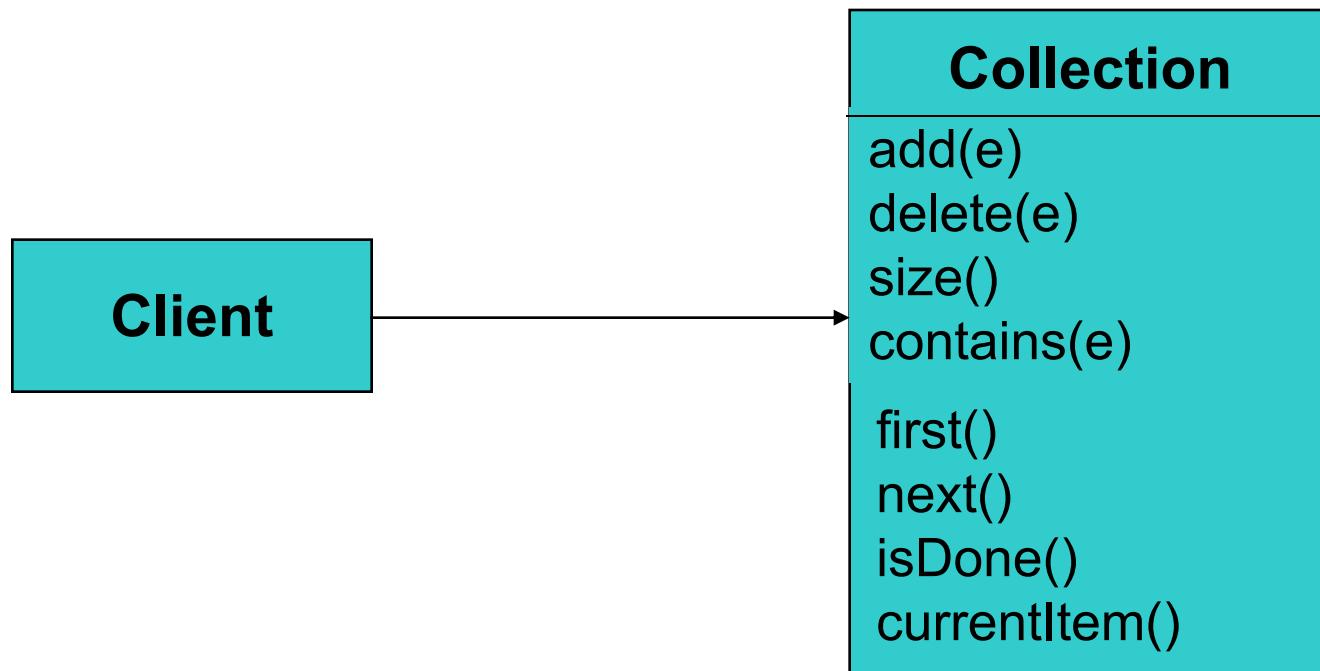
- Access all elements in the collection
- Do not expose internal structure
- Access is independent of the collection
- Multiple accesses can be done independent of each other
- Collection can be maintained independent of the access

○ Solution

- Iterator object controls access

Embedded Iterator

- Embed Iterator support in the Collection



Issues with Embedded Iterators

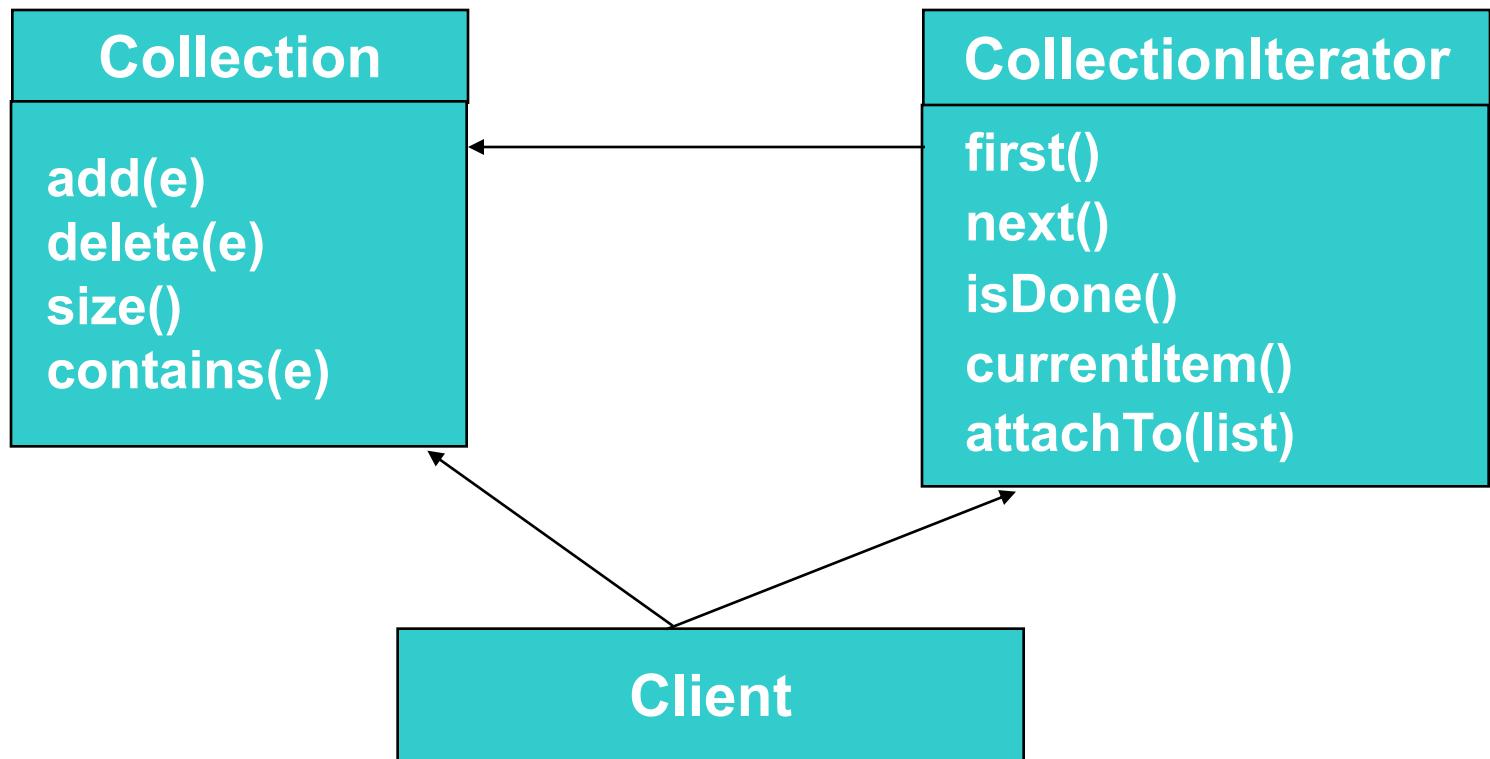
- Mixes structure maintenance and traversal
- Only one traversal at a time
- Adds to collection interface
- Adds to collection implementation
- Clumsy when adding different traversal algorithms:
 - Lists: forward / reverse traversal
 - Trees: pre / post / inorder
 - Graphs: depth first vs. breadth first



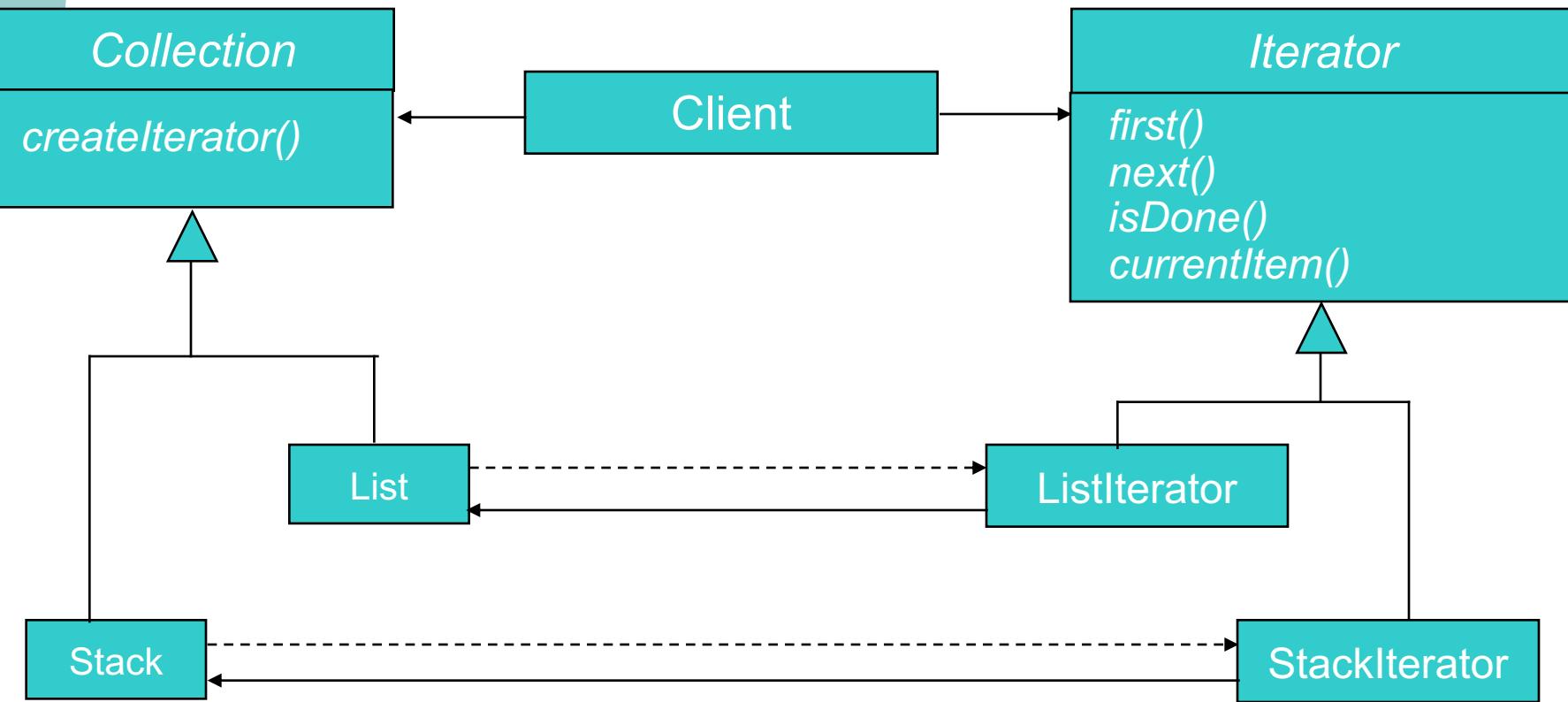
Alternative: Separate Iterator Class & Objects

- Create iterators as needed for a collection
- Attach to collection at creation time or via attach method
- Use iterator object, not collection, to access elements.

Separate Collection Iterator



Abstract Collections & Iterators



Consequences

- Supports various traversal algorithms via different iterators over same collection.
- Simplifies collection interface: At most must return different iterators
- More than one traversal can be in progress

Internal and External Iterators

- Examples so far are external iterators
 - Client object does iteration using provided methods
- Internal iterators
 - The iterator controls the traversal.
 - Client hands iterator some operation to perform on each element in aggregate.
 - Iterator traverses aggregate (Iteration done by iterator object itself)

Builder pattern

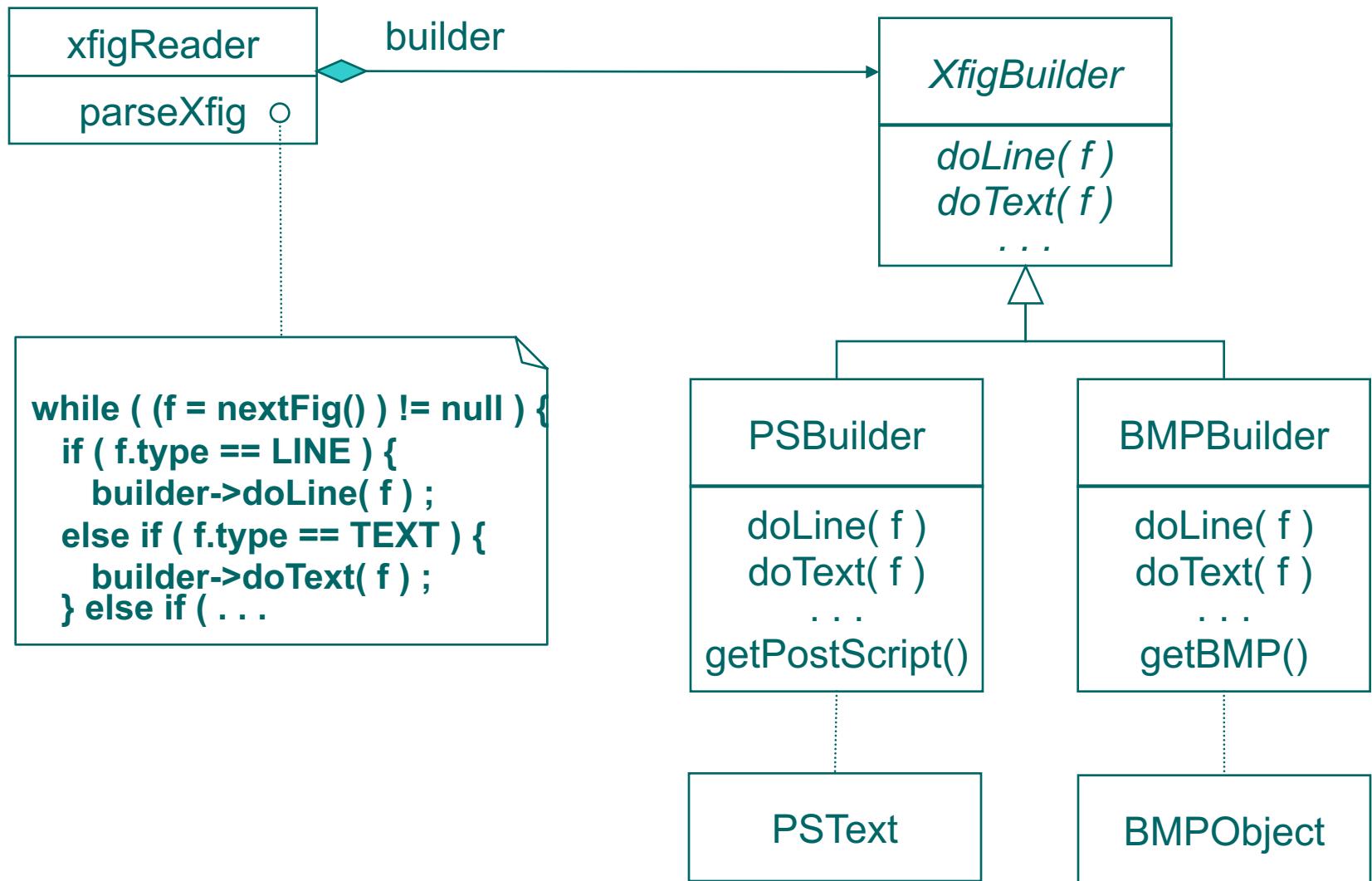
The Problem

- Need to construct complex object incrementally.
- There may be many variants of complex object
- Example: *fig2dev*
 - One input -- the *xfig* file of figure descriptions
 - Many output objects:
 - Postscript
 - BMP for Windows
 - TIFF for fax
- **fig2dev -L language [fig-file [out-file]]**
- How to abstract the construction process?

Possible Solution: Builder

- Builder provides abstract interface for incremental construction
- Concrete builders provide variant on this interface
- Select desired concrete builder and provide to client
- Client simply invokes interface methods for each constituent "piece"
- Parameterizes the construction of many variants.

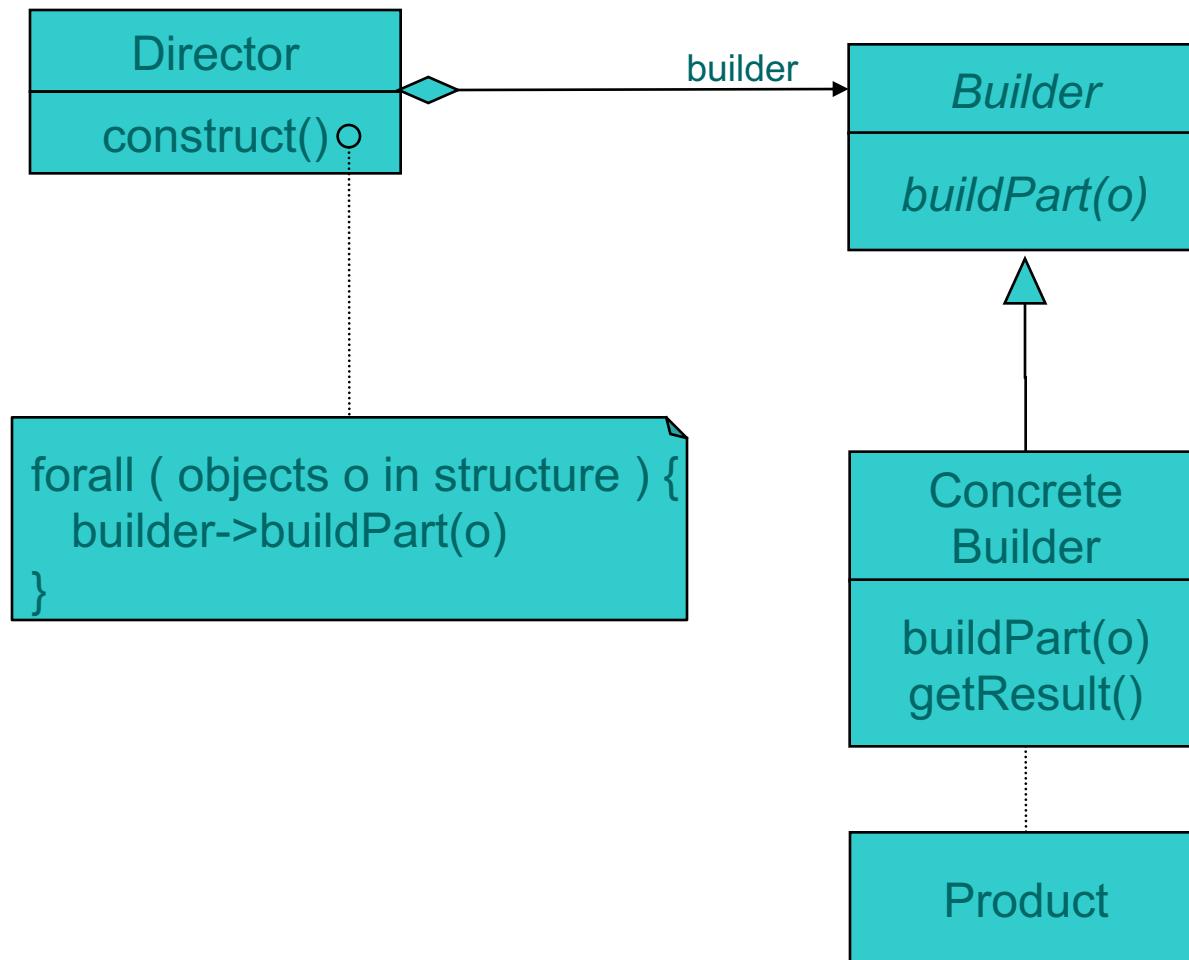
A Builder for Xfig Format



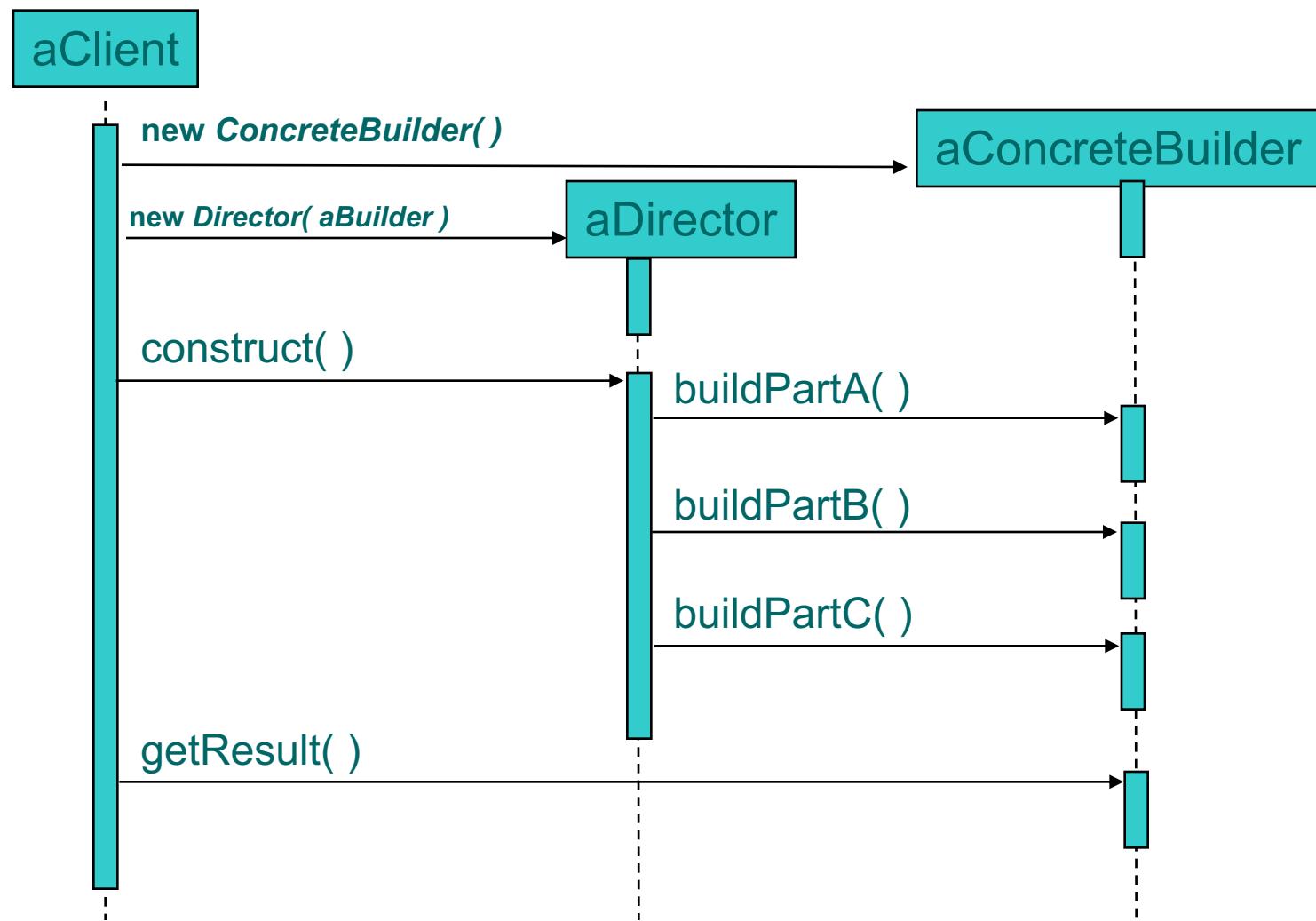
Use the Builder Pattern When

- Need algorithm / assembly independence
 - Algorithm for construction in the client
 - Specific parts of complex object hidden in builder
 - Assembly performed in builder
- Many variant representations
 - Construction fixed
 - Construction incremental
 - Many possible construction variants

General Builder Structure



Builder Interaction Diagram



Collaborations

- *Client* creates *Director* and configures it with a specific *Builder*
- *Director* notifies *Builder* when a part is to be added to the *Product*
- *Builder* adds parts requested to the *Product*
- *Builder* returns *Product* to *Client* on request

Consequence: Can Vary Product

- Can add new types of products built according to common process
- In our example, can add drivers for other forms of xfig output.
- Specifics of how assembly proceeds are hidden, as long as every legal assembly sequence can be accommodated

Consequence: Isolates Construction Algorithm From Representations

- Encapsulates *process* of construction in *Director*
 - Might construct objects from different inputs
 - Example: fig2dev driven by another drawing program format
- Encapsulates *product* of construction in *Builder*
- Can vary each independently as long as *Builder* interface is stable.
- Note: There may be semantic constraints on *Builder* interface sequencing:
 - E.g. each "startComposite()" needs a matching "endComposite()"

Consequence: Fine-Grained Construction Control

- Does not construct full product object in one-shot
- Supports step-by-step construction: Like building a house
- Only retrieve the product when Director is finished with the construction
- Changes to *Directors*
 - Must recognize new types of parts
 - Must know how to invoke construction of part inside the Builder
- Changes to *Builders*
 - Need to add a new part construction method to interface
 - Need to update all subclasses (e.g., all concrete builders)

Command pattern

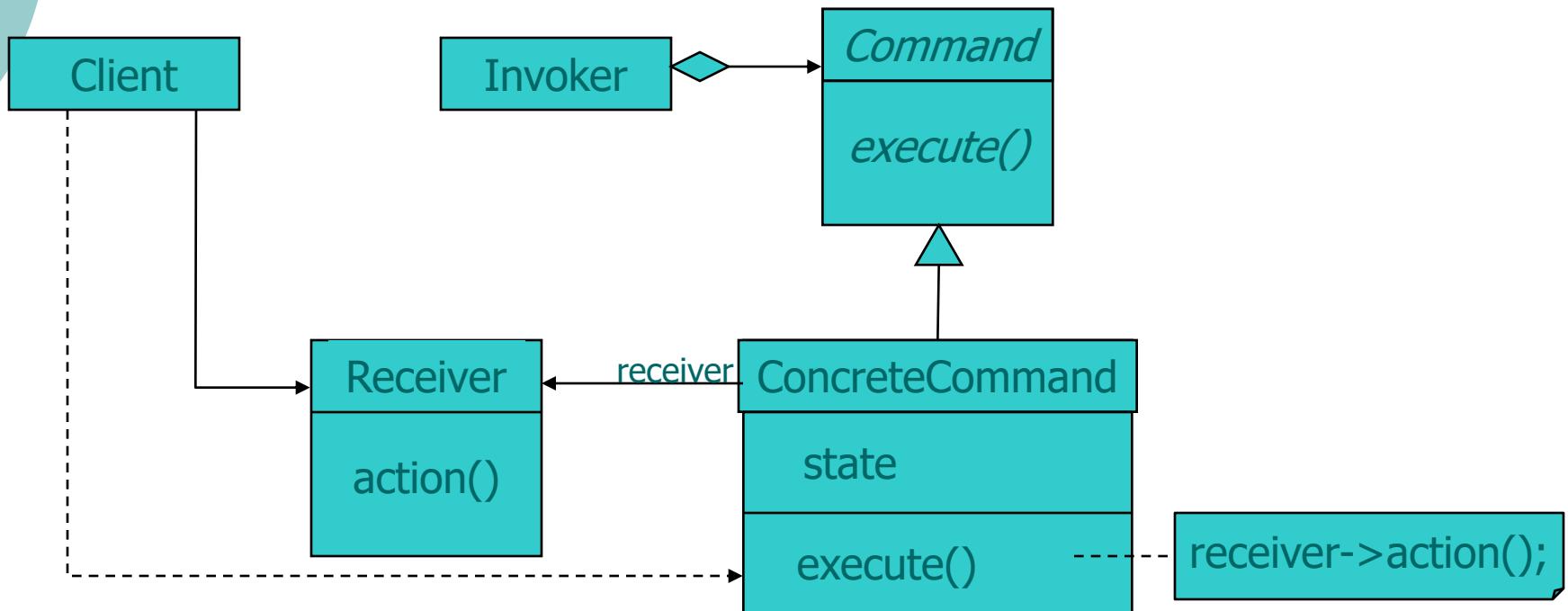
The Problem

- Problem
 - An application has multiple commands being issued via different mechanisms
- Desire: Decouple invoking of the action from
 - Knowledge of how to perform it
 - Knowledge of the receiver of request
- Solution
 - Create **Command** objects that know how to execute the operations

Example

- Image manipulation program allows commands to be issued from
 - Toolbar item
 - Menu item
 - Script language
 - Fly-over menu
- All these interface elements should not need to know how to perform the operation

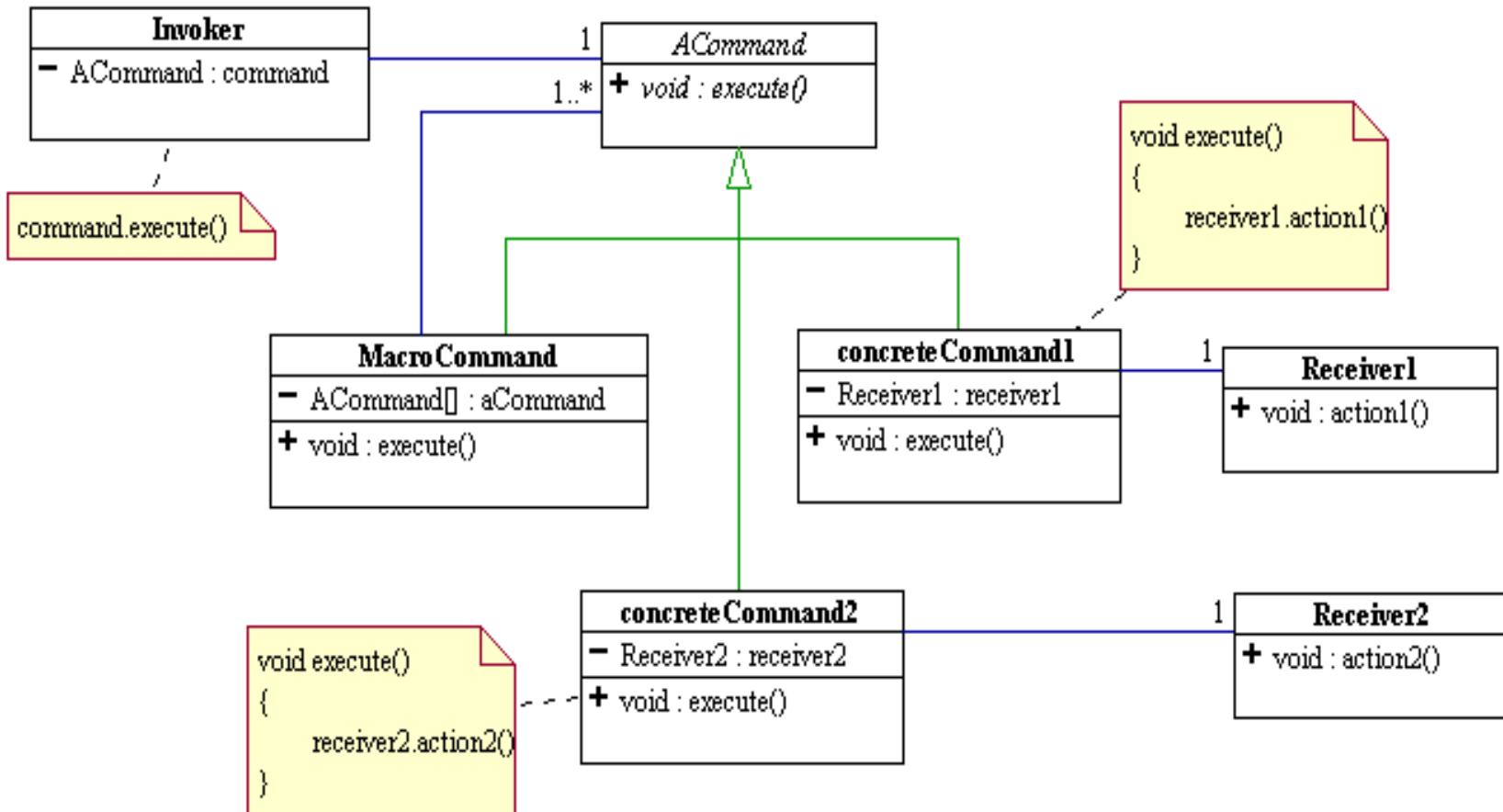
Pattern Structure



Participants

- **Command**
 - Interface for executing every operation
- **Concrete Command**
 - Implements operation
 - Binds receiver and action
- **Client**
 - Creates Concrete Command
 - Determines Receiver
- **Invoker**
 - Requests command to execute operation
- **Receiver**
 - Performs the operations needed

Example



Consequences

- Invocation is decoupled from execution
- Command is an object \Rightarrow it can be extended
- Macro commands accommodated
- Easy to add new commands to the application interface

Undo / Redo

- Unexecute operation
- How to return to original state of receiver?
 - Save state in receiver?
 - “Unaction” in receiver?
 - Save state in command? ⇒ **Memento**
- History list
 - Going backward
 - Going forward
 - Do you need a copy of command? ⇒ **Prototype**
- Reliable restoration over repeated undo/redo cycles

Complexity

Lines of Code

- Size correlates with defect density:
 - Small modules have high rates
 - Medium modules have low rates
 - Large modules have high rates
- Many possible measures of size, but LOC is easiest to measure

Halstead's Software Science

- Programming is a process of arranging tokens: operators and operands
- Derived a set of equations that characterize software in terms of those tokens

Software Science Metrics

Vocabulary (n) = $n_1 + n_2$

Length (N) = $N_1 + N_2$

Volume (V) = $N \log_2(n)$

Faults (B) = V / S^*

where $S^* = 3000$ is the mean number of decisions between errors

Example Program

```
int mult(int a, b) {  
    int c, r;  
    if (a < 0) {  
        return 0 }  
    r = 0;  
    c = a;  
    while (c > 0) {  
        r = r + b;  
        c = c - 1; }  
    return r;  
}
```

Operators

```
int mult(int a, b) {  
    int c, r;  
    if (a < 0) {  
        return 0 }  
  
    r = 0;  
    c = a;  
    while (c > 0) {  
        r = r + b;  
        c = c - 1; }  
  
    return r;  
}
```

Operators:

n1 = 9

N1 = 15

Operands

```
int mult(int a, b) {  
    int c, r;  
    if (a < 0) {  
        return 0 }  
  
    r = 0;  
    c = a;  
    while (c > 0) {  
        r = r + b;  
        c = c - 1; }  
  
    return r;  
}
```

Operators:

n1 = 9

N1 = 15

Operands:

n2 = 7

N2 = 21

Example Software Science Metrics

$$\begin{aligned}\text{Vocabulary (n)} &= n_1 + n_2 \\ &= 9 + 7 \\ &= 16\end{aligned}$$

Operators:
 $n_1 = 9$
 $N_1 = 15$

$$\begin{aligned}\text{Length (N)} &= N_1 + N_2 \\ &= 15 + 21 \\ &= 36\end{aligned}$$

Operands:
 $n_2 = 7$
 $N_2 = 21$

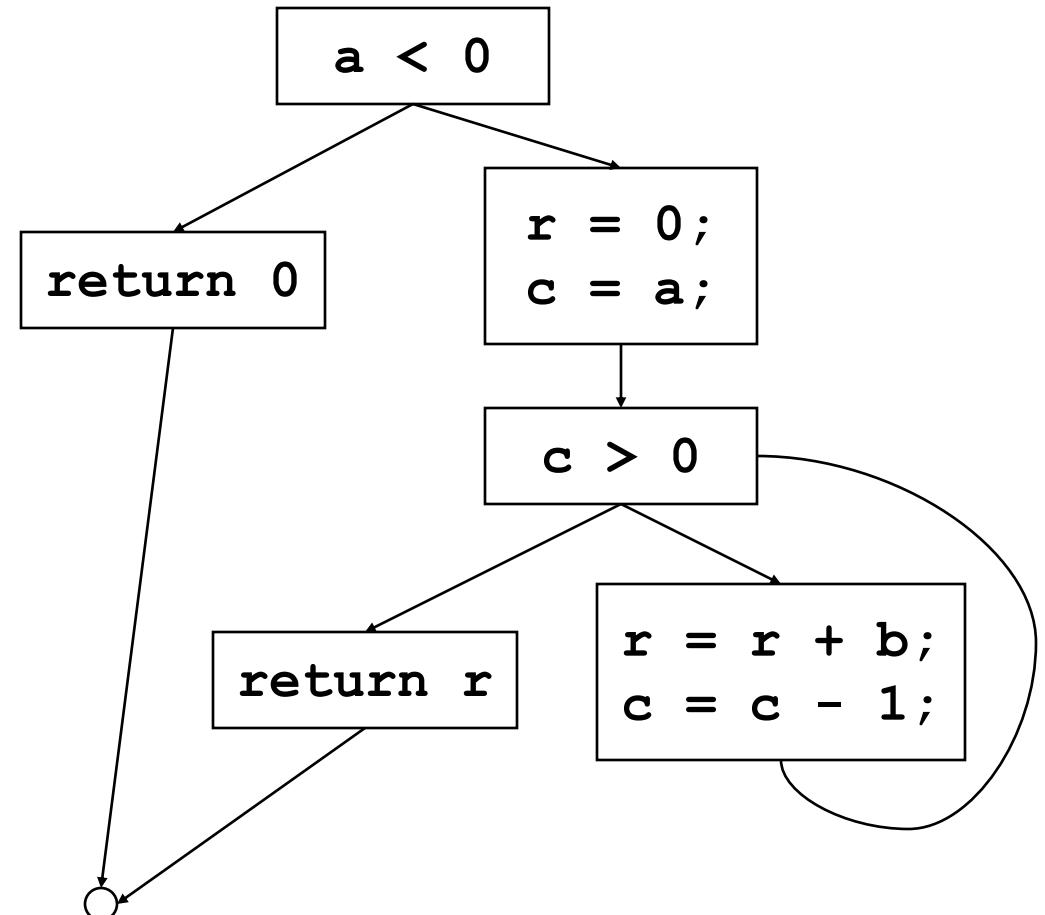
$$\begin{aligned}\text{Volume (V)} &= N \log_2(n) \\ &= 36 * \log_2(16) \\ &= 36 * 4 \\ &= 144\end{aligned}$$

Cyclomatic Complexity

- Measure of control flow complexity
- Sometimes used for testing: it measures the number of branches
- $V(G) = e - n + 2$
 - e = number of edges in flow graph
 - n = number of nodes in flow graph

Example Program Flow Graph

```
int mult(int a, b) {  
    int c, r;  
    if (a < 0) {  
        return 0 }  
    r = 0;  
    c = a;  
    while (c > 0) {  
        r = r + b;  
        c = c - 1; }  
    return r;  
}
```



Example Cyclomatic Complexity

$$e = 8$$

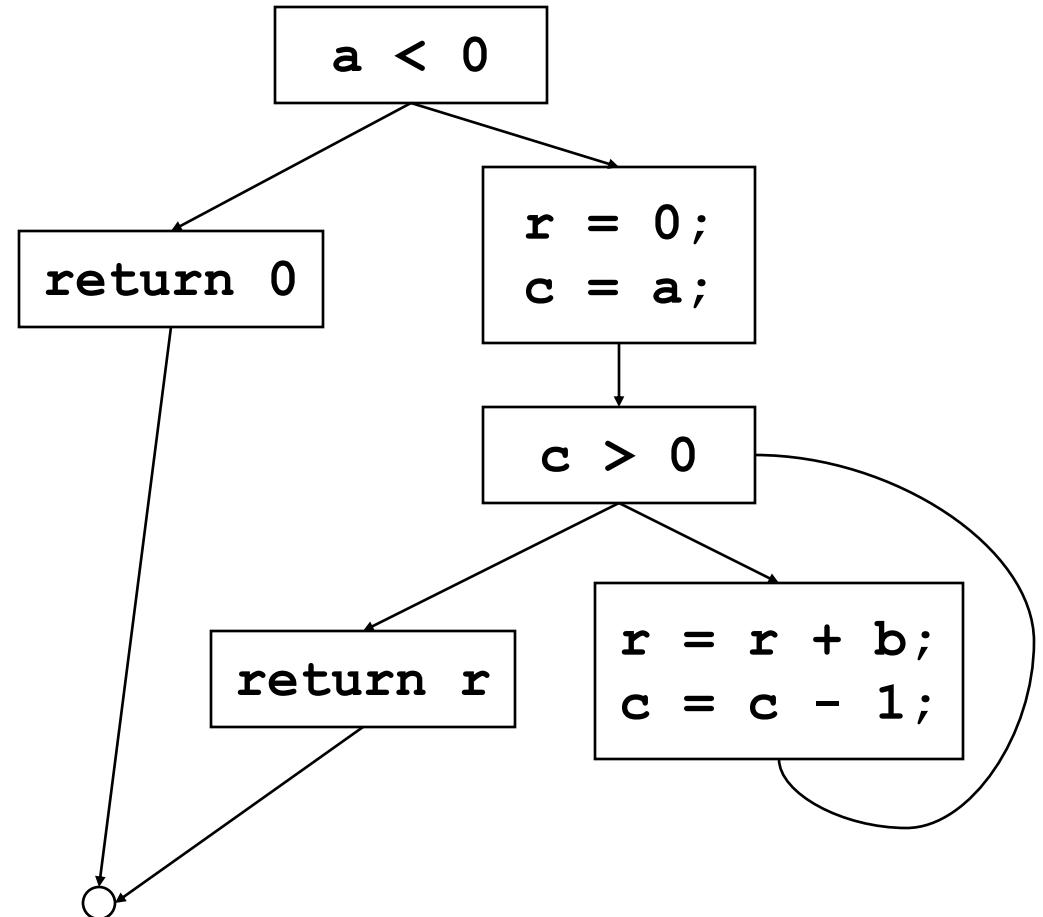
$$n = 7$$

$$p = 1$$

$$V(G) = e - n + 2p$$

$$= 8 - 7 + 2$$

$$= 3$$



System Complexity

- Measure complexity of a system of modules in terms of:
 - Structure: measure of the fan-out of individual modules
 - Data: measure of I/O variables
- Evidence that this correlates with defect density

OO Rules of Thumb (Lorenz)

- Lorenz proposed guidelines for OO development:
 - Average Method Size < 24 LOC
 - Average Methods/Class < 20
 - Depth of Inheritance Tree < 6
- ...
...

CK Metrics

- Chidamber and Kemerer proposed 6 OO metrics (CK metrics):
 - Weighted Methods per Class
 - Depth of Inheritance Tree
 - Number of Children of a Class
 - Coupling Between Object Classes
 - Response for a Class
 - Lack of Cohesion on Methods

Weighted Methods per Class

- Sum of the cyclomatic complexities of methods
- Number of methods is often used instead

Depth of Inheritance Tree

- Length of the maximum path in the class hierarchy
- Represents the number of opportunities to override methods

Number of Children of a Class

- Number of immediate subclasses of a class
- Measures width of the inheritance tree

Coupling Between Object Classes

- Number of classes whose methods or instance variables are called/referenced
- Some languages (e.g., Smalltalk) encourage higher values

Response for a Class

- Number of methods that can be executed in response to a message received by an object of the class
- Related to response set of a class = all methods called by local methods

Lack of Cohesion on Methods

- Number of disjoint sets of variables used by local methods
- Measures dissimilarity between methods

SOFTWARE MEASUREMENT FUNDAMENTALS

Why Measure?

- Functionality is sacrosanct!!!
 - Quality of the solutions differentiate one from another. We have to turn our solutions about quality into measurables.
- Logical or mathematical, orderly and reliable representation of observation
- Relate reality to data

Basics Elements

- Metrics – a Quantifiable characteristic of an entity
- Measurement - the process of mapping from real world attributes to a mathematical representation
- Models – A Mathematical relation between metrics
 - Ex: between quality attributes and available metrics
- Validity – Does the metric / model accurately represent / measure what it purports to
- Prediction System – set of attributes / model used to predict some futuristic attribute of an entity
 - Deterministic – same result for same inputs
 - Stochastic – provides a window of error around actual measurement

Example Metrics

- Number of Static Methods
- Lines of Code
- Afferent Coupling
- Efferent Coupling
- Normalized distance
- Number of Classes
- Specialization index
- Instability
- Number of Attributes
- Number of Packages
- Weighted methods per class
- Number of overridden methods
- Nested Block Depth
- Number of methods
- Lack of Cohesion Methods
- Cyclomatic complexity
- Number of parameters
- Abstractness
- Number of Interfaces
- Depth of Inheritance Tree

Quality & Metrics

The Quality Concepts
(abstract notions of quality properties)

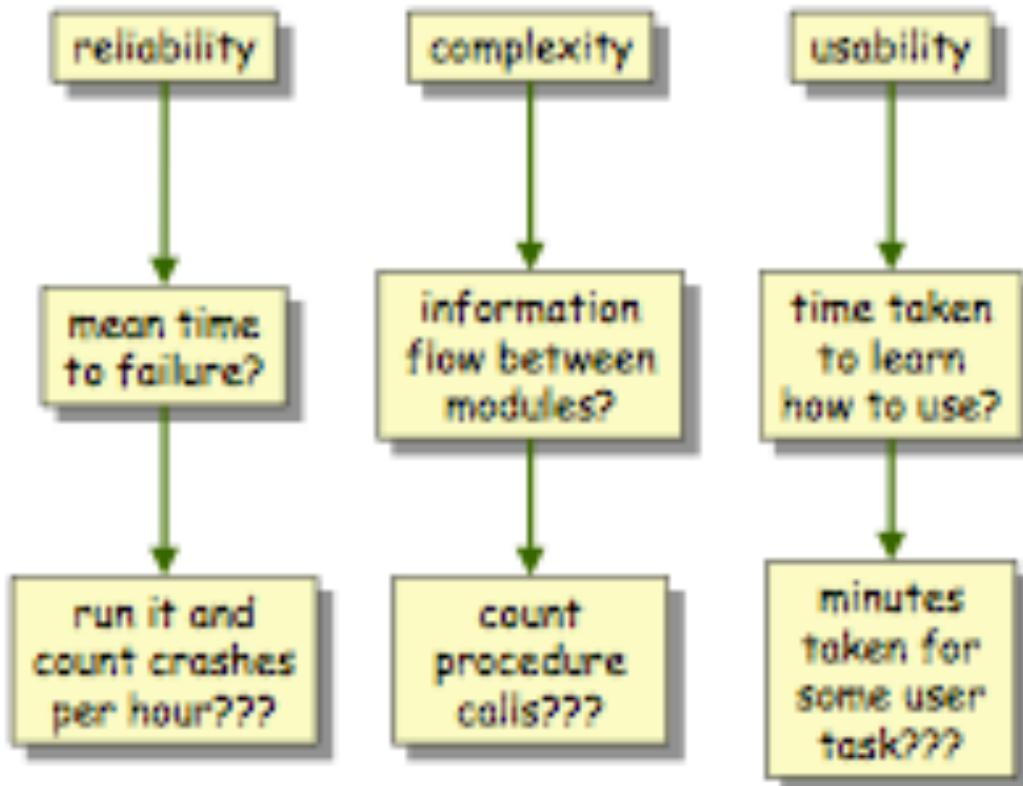


Measurable Quantities
(define some metrics)



Counts taken from Design Representations
(realization of the metrics)

examples...



Examples

- Measure Reliability through
 - Mean time to Failure
 - Mean Time between Failures
- Defect Rates/ Density
 - .002 ppm
- Modularity
 - Cohesion
 - Coupling
- Simplicity
 - Complexity of control / information flow / Name Space

Another Example

Function Points

- ↳ used to calculate size of software from a statement of the problem
- ↳ tries to address variability in lines of code estimates used in models such as COCOMO
 - > e.g. because SLOC varies with different languages
- ↳ Originally for information systems, although other variants exist
- ↳ Basic model is:
$$FP = a_1I + a_2O + a_3E + a_4L + a_5F$$

metric from problem statement
weighting factor for this metric

Example

- ↳ Sets of weightings (a_i) provided for different types of project
- ↳ Measure properties of the problem statement:
 - > I = number of user inputs (data entry)
 - > O = number of user outputs (reports, screens, error messages)
 - > E = number of user queries
 - > L = number of files
 - > F = number of external interfaces (to other devices, systems)
- ↳ Example calculation:
$$FP = 4I + 5O + 4E + 10L + 7F$$

Measurement

- Components of a measurement system:
 - $m = \langle \text{attribute}, \text{scale}, \text{unit} \rangle$
- Can be Deterministic or Probabilistic
- Can be Qualitative or Quantitative
- Can be Direct Measurement or Indirect Measurement
 - Direct: Software Length = n KLOC
 - Indirect: Productivity - # of LOC produced per hour
 - What about quality?

Levels of Measurement

- Scale : an abstract measurement tool to represent a state or attribute of an entity
- Scale Types
 - Objective
 - Nominal Scale
 - Ordinal Scale
 - Internal Scale
 - Ratio Scale
 - Subjective
 - Likert Type

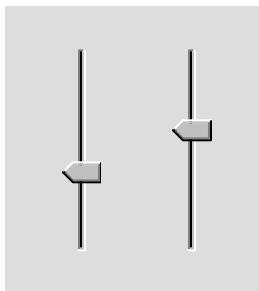
- Desired characteristics of:
 - Distinctiveness: $A \neq B$, $A \neq B$.
 - Ordering in magnitude: $A < B$, $A = B$, $A > B$.
 - Equal/unequal intervals: $A-B < C-D$, $A-B = C-D$,
 - $A-B > C-D$.
 - Ratio: $A \square k B$ (absolute zero is required).
 - Absolute magnitude: $A = k_a \text{ REF}$, $B = k_b \text{ REF}$ (absolute reference or unit is required).

Nominal Scale

- Refers to measurement based on classification. For example,
 - Process can be categorized as Waterfall, Spiral, Iterative, etc.
 - Continents as Asia, America, Africa, Europe, etc.
 - Faults as Specification fault, Design fault or coding fault
- No notion of ordering, magnitude or relationship between the categories

Ordinal Scale

- Refers to measurement based on classification where the elements can be compared in order. For example,
 - People can classified according to socio economic status: upper class, upper-middle class, middle class, lower-middle class, and lower class
 - SEI CMM maturity levels
 - Complexity Levels (trivial, Simple, Moderate, Complex, Very complex)
- It's symmetric and transitive
- Does not offer any information about the magnitude of the difference between the elements



Examples:
IQ test,
competition
results,
etc.

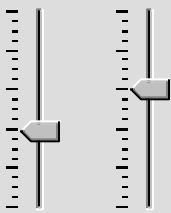
State \equiv order

Image₁

		$A = 1$ $B = 1$	$A = 2$ $B = 1$
		$A = 2$ $B = 1$	$A = 1$ $B = 2$

Interval and Ratio Scale

- Interval scale indicates the exact differences between measurement points
- Captures information about the size of the intervals that separate the classes
- An interval scale preserves order, as with an ordinal scale.
- If an absolute or non-arbitrary zero point can be located on an interval scale, it becomes a ratio scale
- A higher level measurement can always be reduced to a lower one but vice versa is not possible.



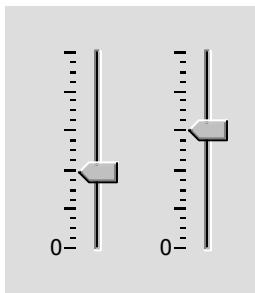
Examples:
time scales,
temperature
scales, etc.,
where the
origin or zero
is not fixed
(floating).

State \equiv interval

$$Image_2 = 10 \times Image_1 + 2$$

A vertical scale with a green arrow pointing upwards from a point below the first tick mark. Two pink arrows point upwards from the second and third tick marks.	A vertical scale with a green arrow pointing upwards from the first tick mark. Two pink arrows point upwards from the second and third tick marks.	$A = 42$ $B = 42$ $ A - B = 0$	$A = 52$ $B = 42$ $ A - B = 10$
A vertical scale with a green arrow pointing upwards from the first tick mark. One pink arrow points upwards from the second tick mark.	A vertical scale with a green arrow pointing upwards from the first tick mark. One pink arrow points upwards from the second tick mark.	$A = 82$ $B = 42$ $ A - B = 40$	$A = 62$ $B = 72$ $ A - B = 10$

Any increasing linear transformation can be used to change the scale.



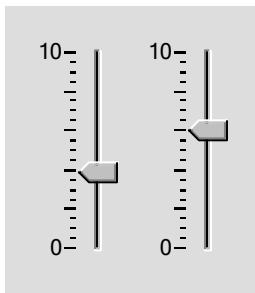
Examples:
measurement
of any physical
quantities
having **fixed**
(absolute)
origin.

State \equiv ratio

$Image_2 = 10 \times Image_1$

		$A = 40$ $B = 40$ $A/B = 1$	$A = 50$ $B = 40$ $A/B = 5/4$
		$A = 80$ $B = 40$ $A/B = 2$	$A = 60$ $B = 70$ $A/B = 6/7$

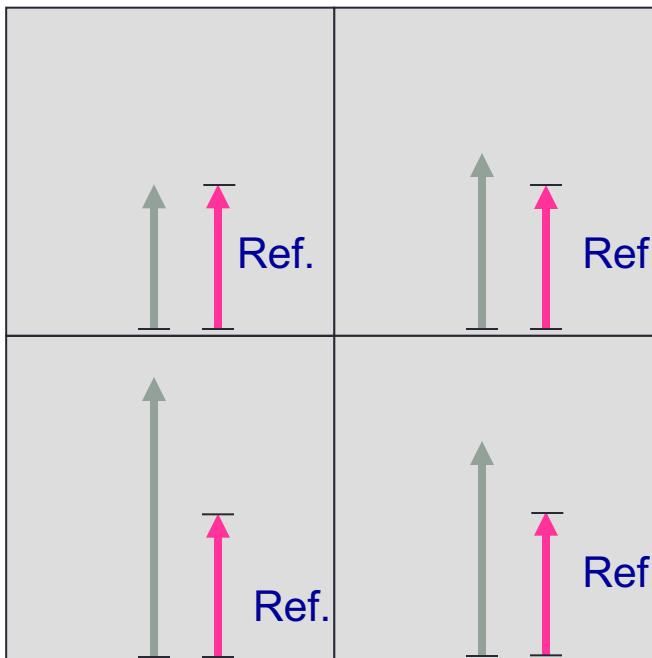
The only transformation that can be used to change the scale is the multiplication by any positive real number.



5. Absolute scale

Examples:
measurement
of any physical
quantities by
comparison
against an
absolute unit
(reference).

State \equiv absolute value



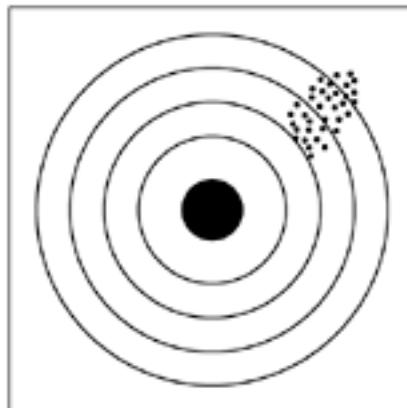
Image

$A = 1$	$A = 5/4$
$A = 2$	$A = 3/2$

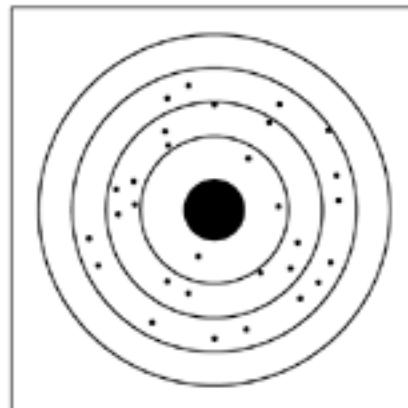
No transformation can be used to change the scale

Reliability and Validity

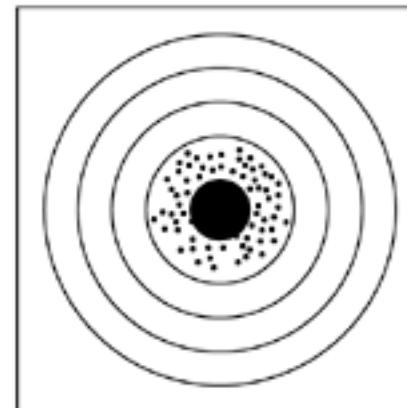
- Reliability: consistency
 - example: a scale that gives the same reading for each measurement
 - Index of variation (IV) measure is used
 - $IV = \text{Standard Deviation} / \text{Mean}$
- Validity: accuracy
 - example: a scale that gives a reading that is close to the real weight



Reliable but not valid



Valid but not reliable



Valid and reliable

Reliability

- Extent to which an experiment, test, or any measuring procedure yields the same result on repeated trials.
- Extent to agreement of measuring instruments / scale over time.
(Stability)
 - Scale does not give different measures of same attribute over time
- extent to which > 2 instruments agree on the measure of identical concepts at an identical level of difficulty (equivalence)
 - Ex: using two difference questions to judge ‘customer satisfaction’ should be equivalent
- extent to which tests or procedures measure the same characteristic, skill or quality (Internal)
 - ALL questions intending to measure customer satisfaction should indeed be related to customer satisfaction.
 - Question: “do you like Pani-puri?” In a customer satisfaction survey is stable, but not internally reliable, since it has nothing to do with customer satisfaction.

Errors

- Systematic Error
 - Validity related error
- Random Error
 - Reliability related error
- Observed Value = Actual Value + Systematic Error + Random Error
- **Reliability = Variance(Actual Values) / Variance(Observed Value)**
- **Reliability = a value between 0 and 1**

Calculating Reliability

- Test –Retest
- Calculate two scores for same attribute
 - Different points in time
 - Different people
- Take correlation between the variance of two scores
- Low correlation = ?
- High correlation =



More Patterns...



Chain of Responsibility

Acknowledgement: Head-first design patterns

Problem

Scenario: *Paramount Pictures* has been getting more email than they can handle since the release of the Java-powered Ironman game. From their analysis they get four kinds of email:

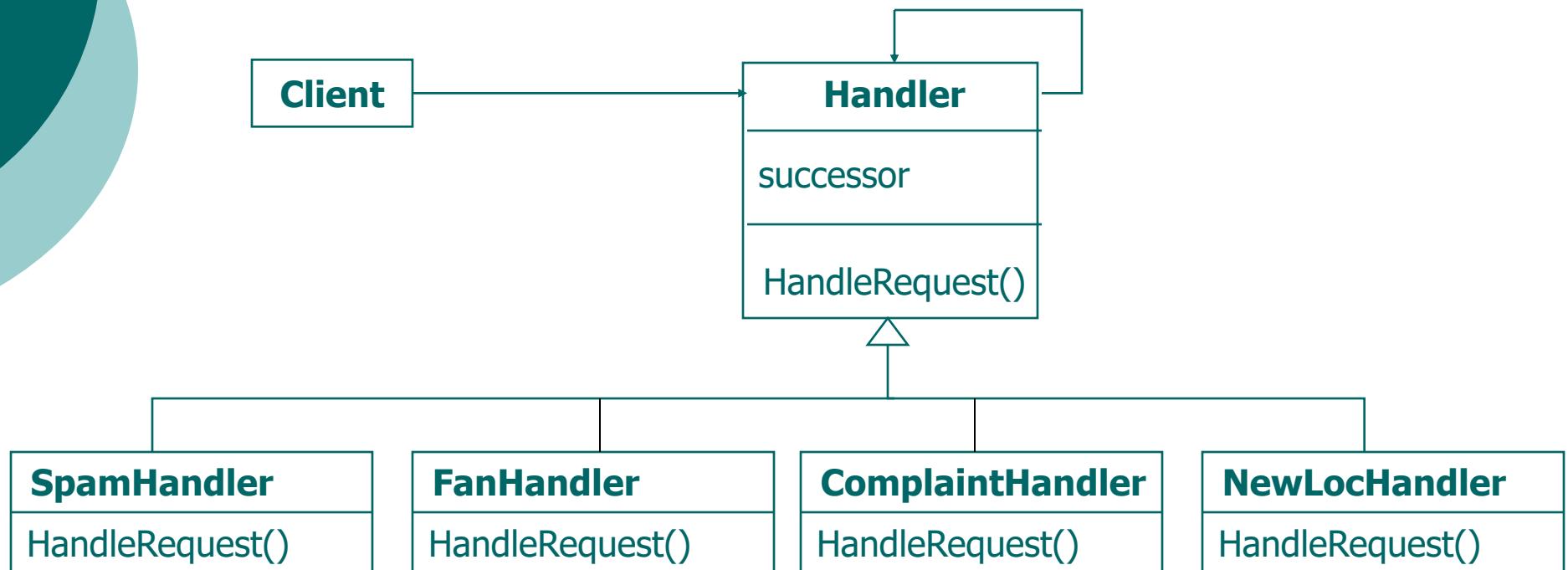
- Fan mail from customers that love the new 1 in 10 game
- Complaints from parents whose kids are addicted to the game
- Requests to put machines in new locations
- Spam

Task: They need you to create a design that can use the detectors to handle incoming email.

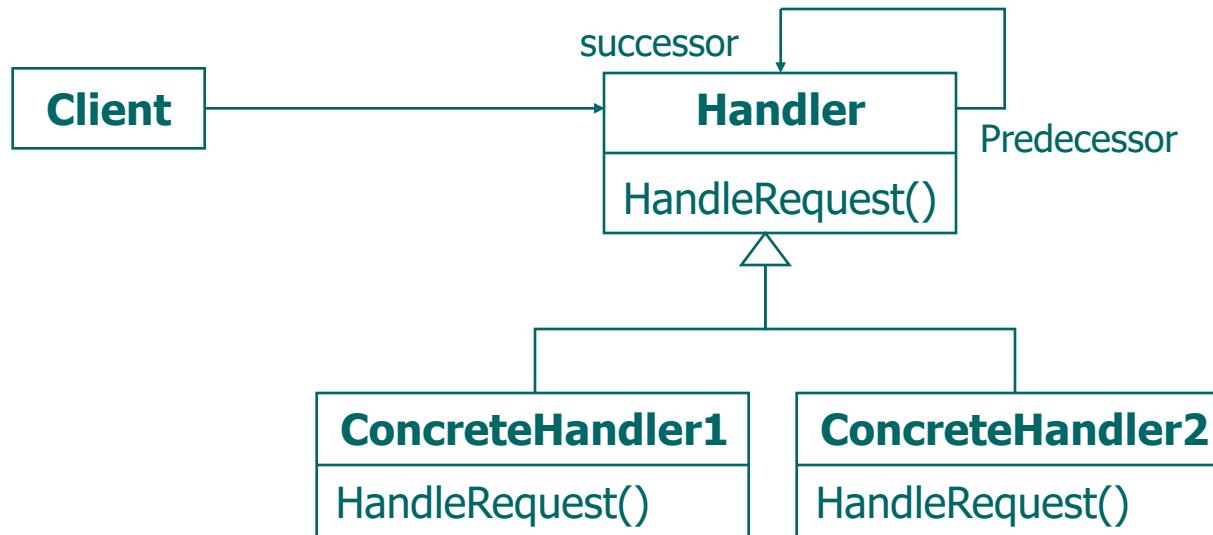
Chain of Responsibility

- Create a chain of objects that examine a request
- Each object examines the request and handles it, or passes it on to the next object in the chain.

Chain of Responsibility – Email Handler



Chain of Responsibility - Structure



Participants

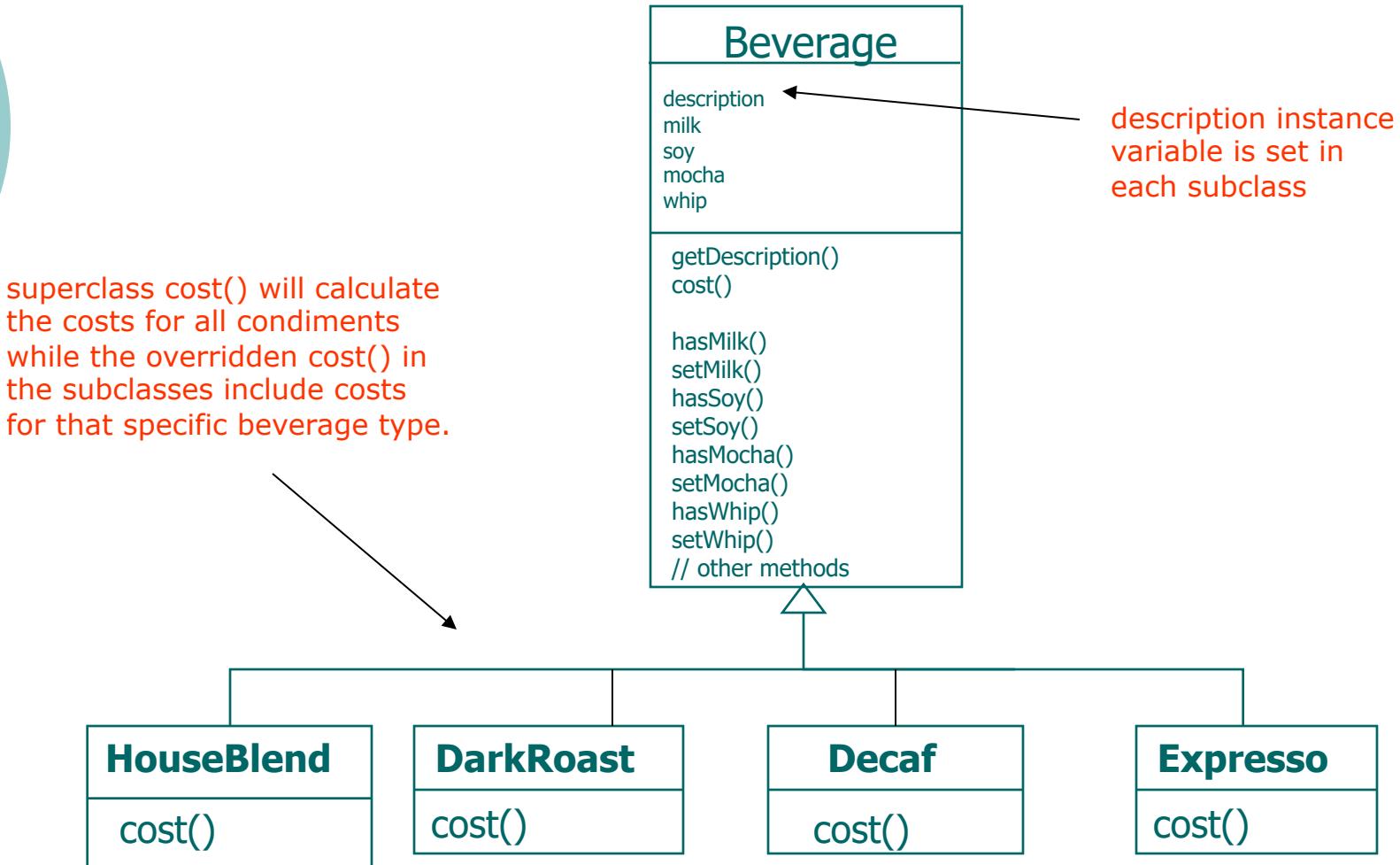
- Handler
 - Defines interface for handling request
 - Implements successor link
- ConcreteHandler
 - Handles all requests for which it is responsible
 - Has access to successor
 - Delegates request to successor when not handled directly
- Client
 - Initiates request to a ConcreteHandler

Benefits and Drawbacks

- Decouples the sender of the request and its receivers
- Simplifies the object because it doesn't have to know the chain's entire structure
- Allows for adding or removing responsibilities dynamically by changing the members or order of the chain
- Commonly used in windows systems to handle events like mouse clicks and keyboard events
- Execution of a request isn't guaranteed
- Can be hard to observe the runtime characteristics and debug

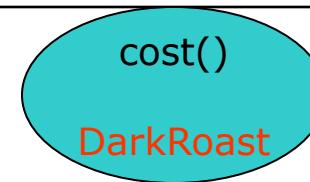
Decorator Pattern

Example - Starbuzz Coffee



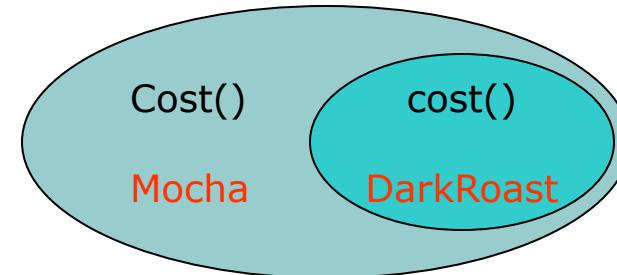
Constructing a drink order with Decorators

- Start with the DarkRoast Object



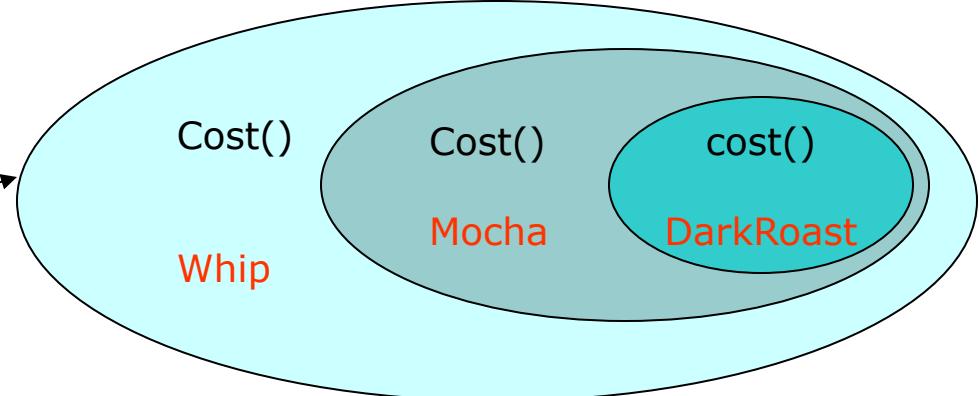
- Customer wants Mocha

Mocha object type mirrors the object it is decorating.



- Customer wants whip

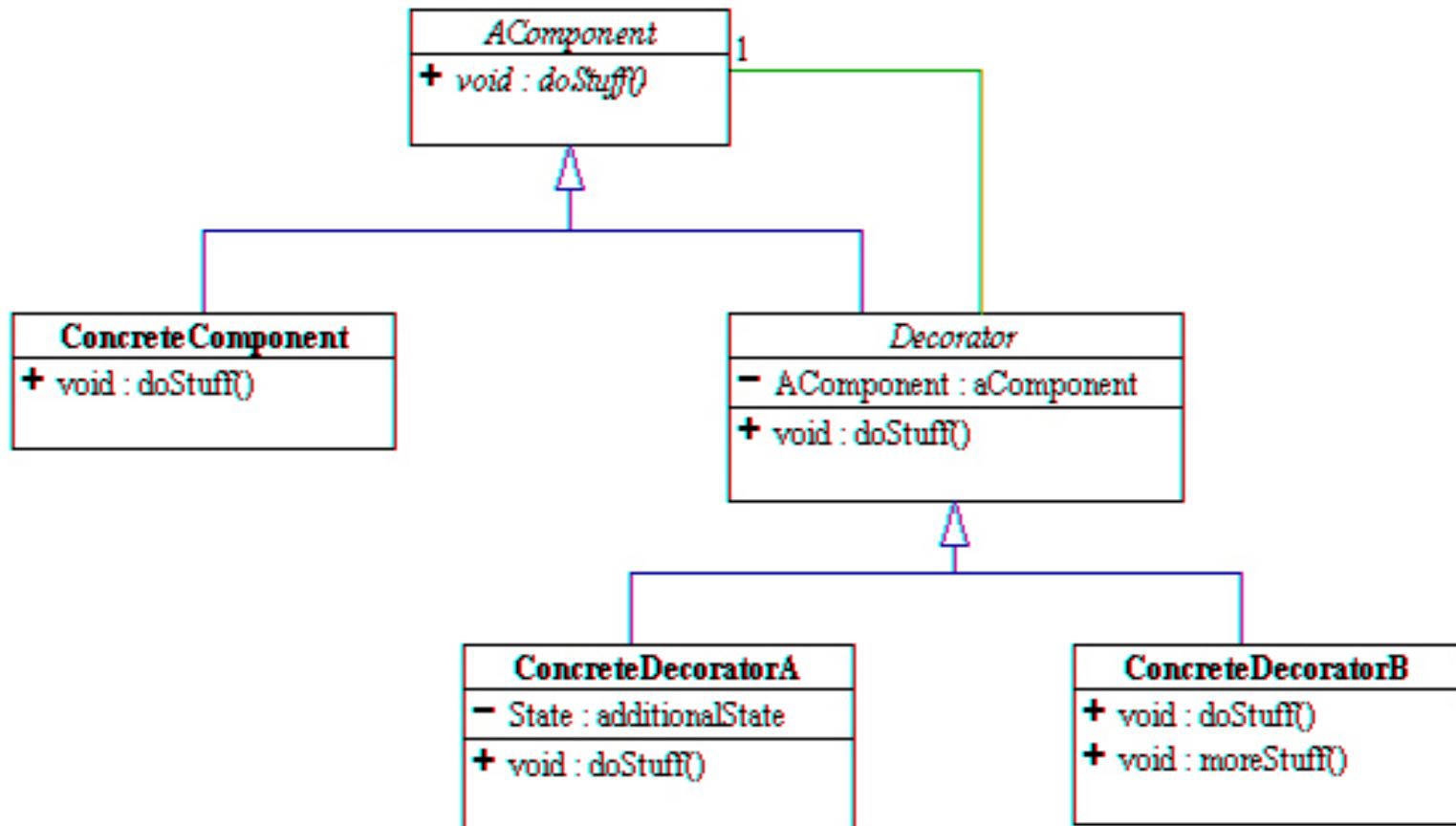
Whip object type mirrors the object it is decorating.



Decorator pattern

- Decorators have the same super type as the objects they decorate
- You can use one or more decorators to wrap an object
- Given that the decorator has the same super type as the object it decorates, we can pass around a decorated object in place of the original object
- **The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job**
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like

General Structure



Mediator Pattern

Problem

Scenario: Bob has an amazing auto-house. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing.

Bob is looking to add additional features:

- No coffee on the weekends...
- Turn off the sprinkler 15 min. before a shower is scheduled...
- Set the alarm early on trash days...

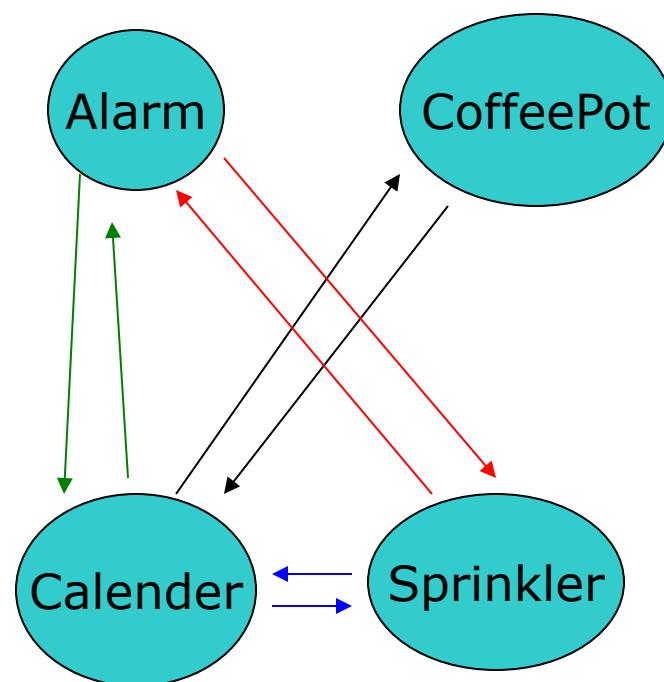
Problem Representation

Alarm

```
onEvent() {  
    checkCalender()  
    checkSprinkler()  
    startCoffee()  
  
    // more stuff  
}
```

Calender

```
onEvent() {  
    checkDayofWeek()  
    doSprinkler()  
    doCoffee()  
    doAlarm()  
  
    // more stuff  
}
```



CoffeePot

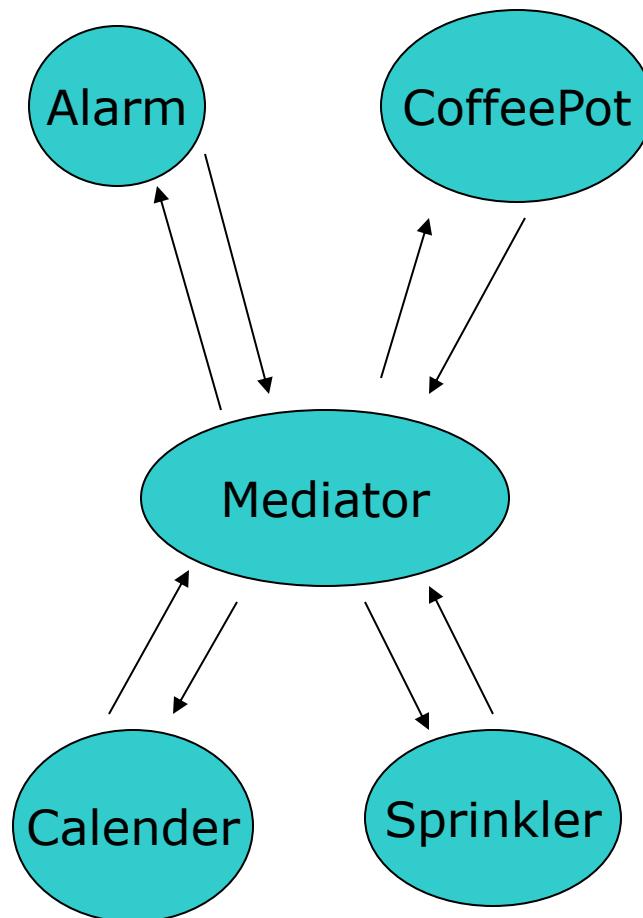
```
onEvent() {  
    checkCalender()  
    checkAlarm()  
    startCoffee()  
  
    // more stuff  
}
```

Alarm

```
onEvent() {  
    checkCalender()  
    checkShower()  
    checkTemp()  
    checkWeather()  
  
    // more stuff  
}
```

Mediator in action...

Mediator
if (alarmEvent) { checkCalender() checkShower() checkTemp() } if (weekend) { checkWeather() // do more stuff } if (trashDay){ resetAlarm() //do more stuff }



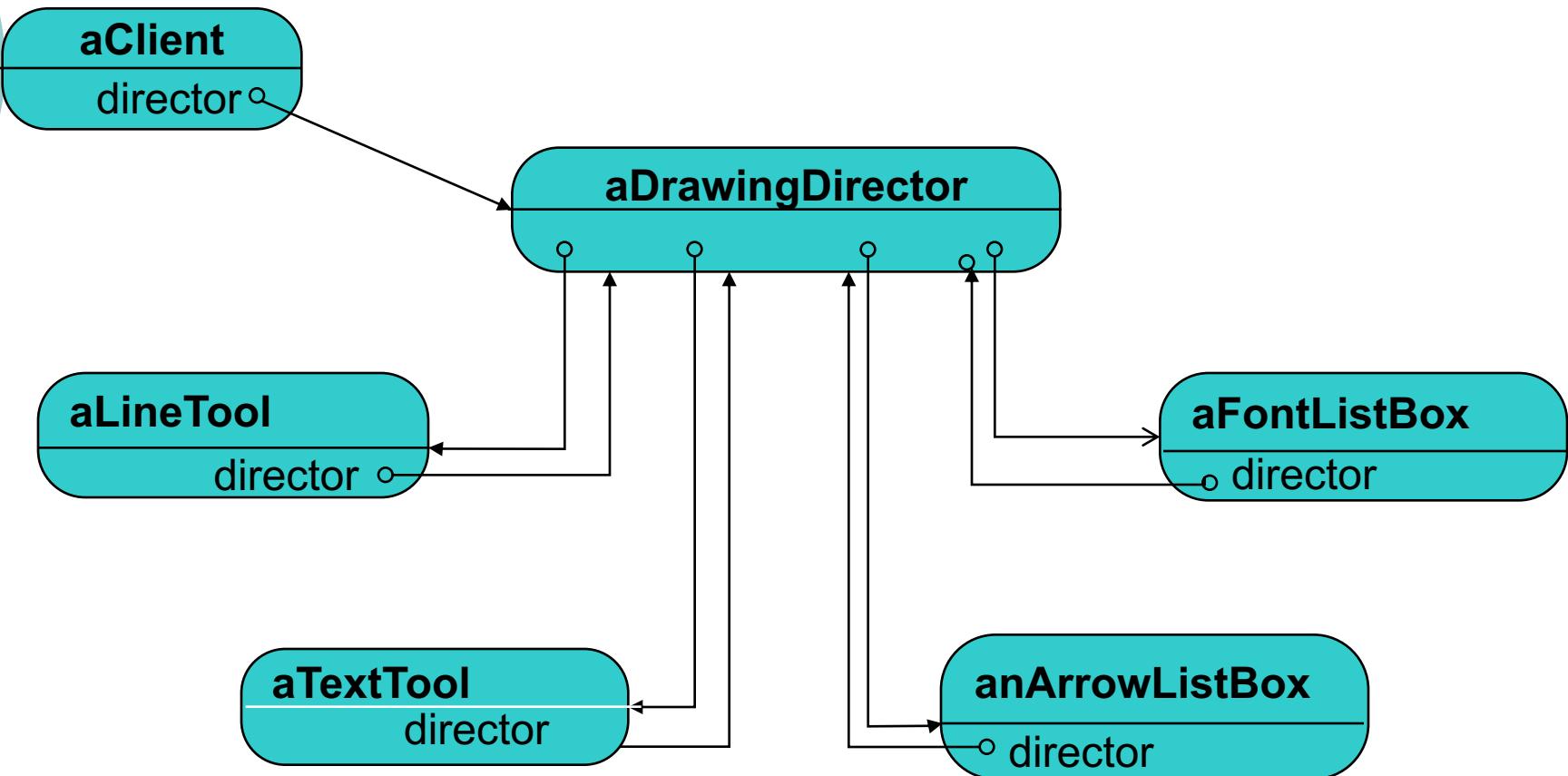
Example: Drawing Editor

- The GUI widgets for a drawing editor are interrelated
- Examples:
 - Insert text enables font widgets and fill widgets
 - Insert 3-D object enables fill-in oriented widgets
 - Insert line or arc enables arrowhead options.
- How to coordinate all these widgets?
- Do not want to have every widget know about every other widget!

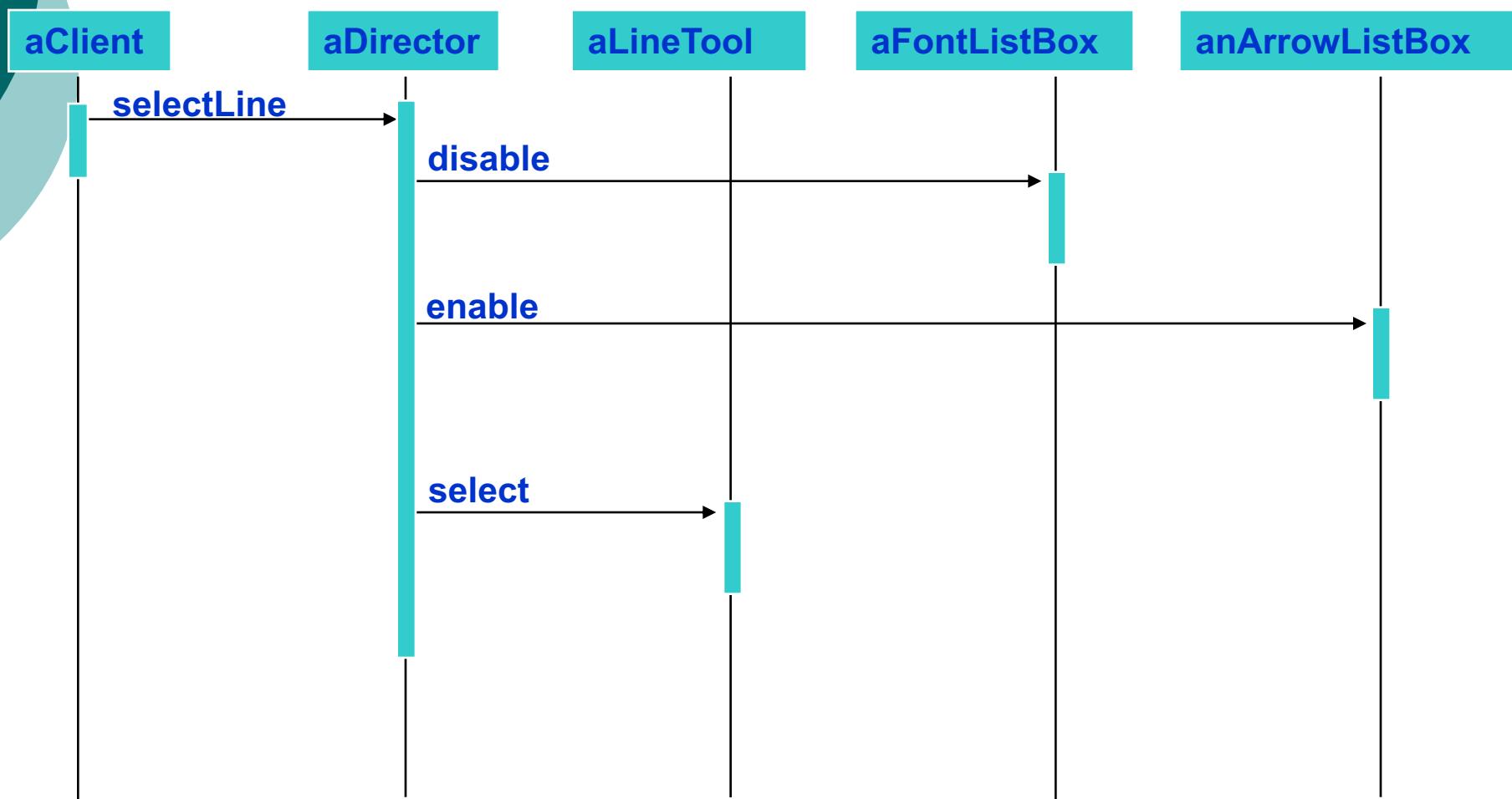
Solution: Mediator Pattern

- Mediator/director objects synchronize object collections.
- Mediator observes all widgets it coordinates
 - Example: for drawing, every change to drawing tool is broadcasted by mediator to other connected widgets
- More efficient and comprehensible than every widget watching every other one.
- Centralized coordination.

Example: Drawing Director



Interaction Chart



Participants

- Mediator

- Interface for communications among colleague objects
- Knows all the colleagues
- Implements cooperative behavior

- Colleagues

- Know their director / mediator
- Communicate with mediator to distribute information to other colleagues

Benefits and Drawbacks

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system
- Simplifies maintenance of the system by centralizing logic
- Simplifies and reduces the variety of messages sent between objects in the system
- Without proper design, the Mediator object itself can become overly complex

Visitor Pattern

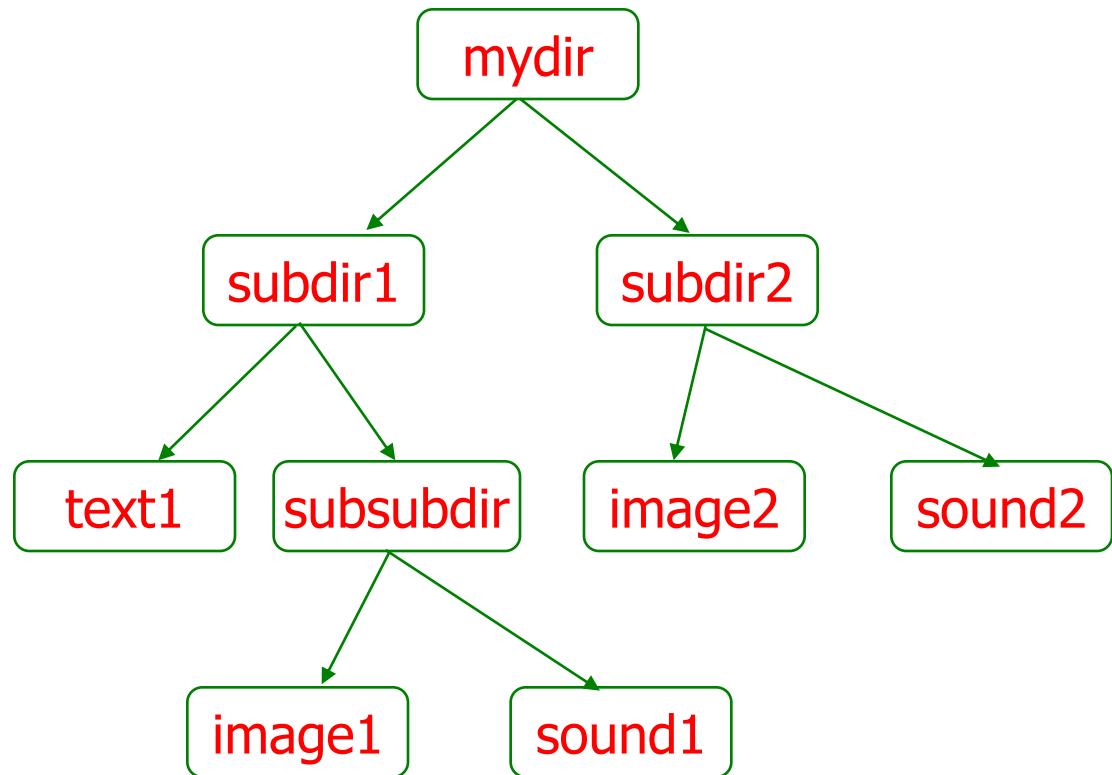
The Problem

- Situation:
 - Operations to be performed on the elements of an object structure
- Desire:
 - Add new operations without changing the classes
 - Perform operation in a type-safe manner

Example – File System

- Basic Types
 - Directories
 - Text Files
 - Image Files
 - Sound Files

- Operations
 - Total size
 - SearchImage



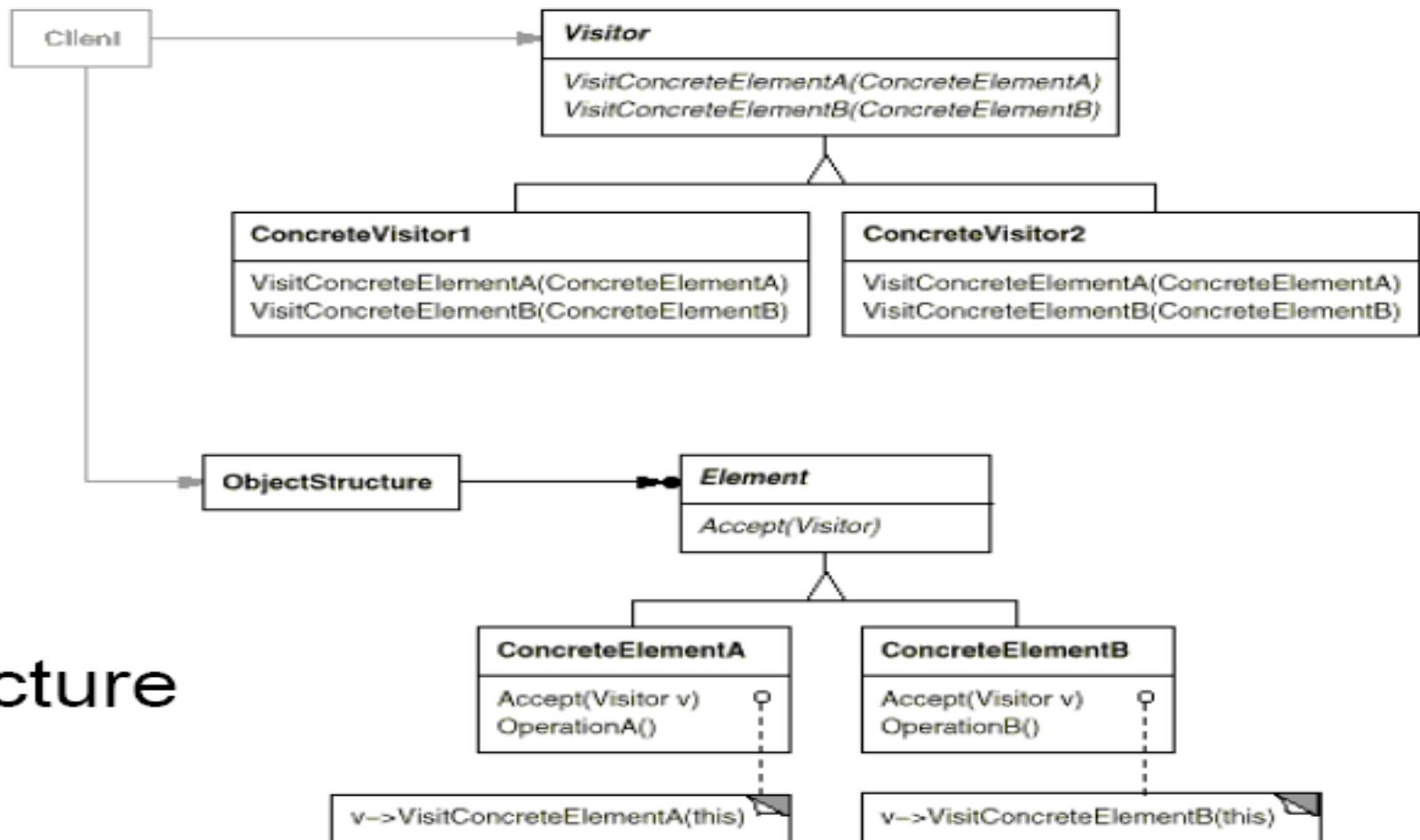
Approach #1 – Operations in Interface

- Define all operations at topmost interface
- Override as necessary in each subclass
- Issue: adding new operations
 - Requires change to top most class
 - Requires *at least* recompilation of all subclasses
 - Typically adds code to each subclass
 - “Method bloat”

Approach #2 - Visitor

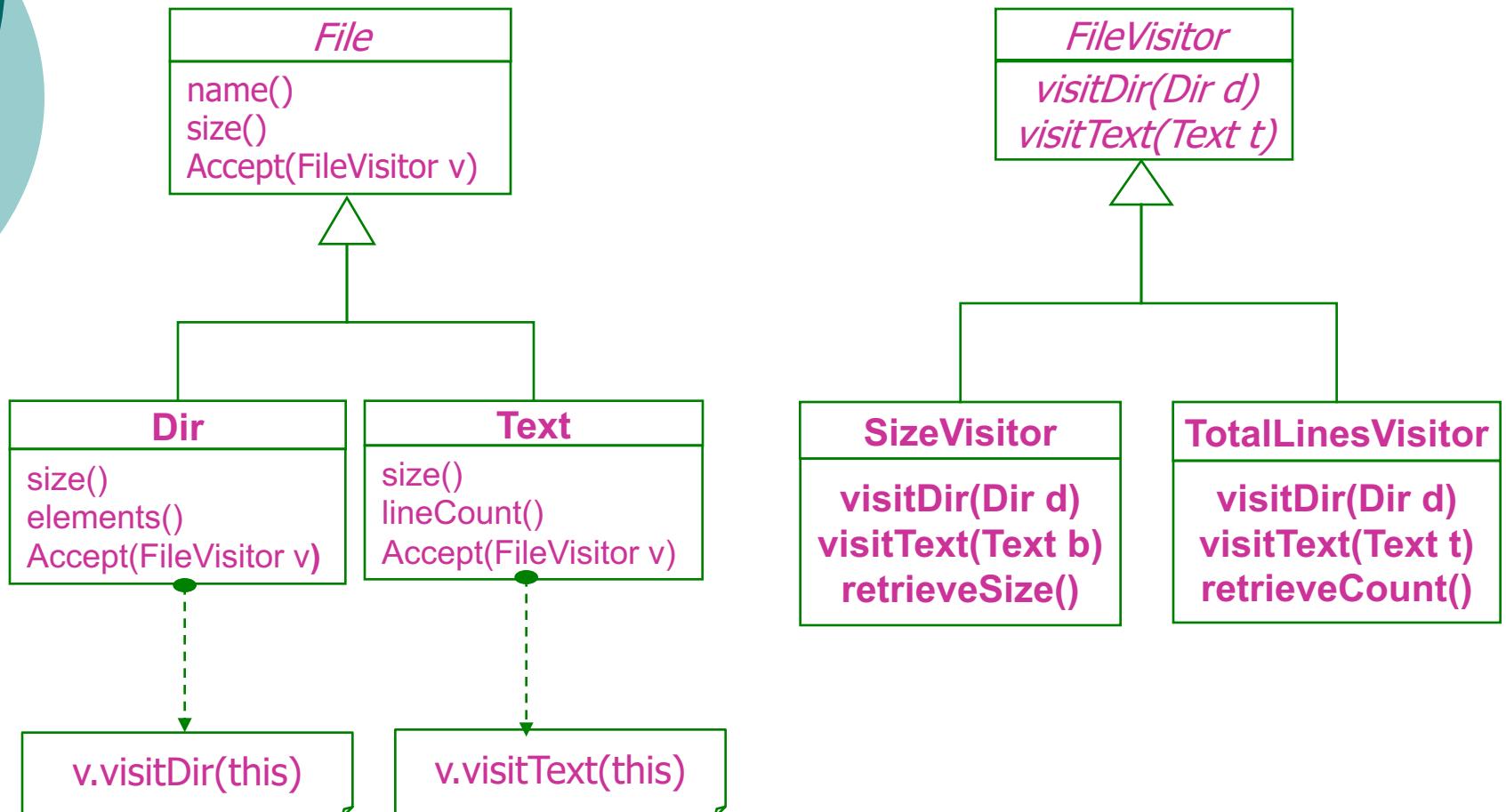
- Define Visitor interface with one method per subclass type (e.g., visitDirectory, visitImage)
- Define **Accept** method in topmost Element interface
- **Accept** method provided a **Visitor** object
- Each class's **Accept** method calls that class's specific method in the **Visitor** object.
- Inside each visitor's method(s) we have access to all functionality of the object being visited

Structure



Structure

Class Diagram



Applicability

- Many distinct, unrelated operations are to be performed
- You want to minimize the common interface
 - Isolates conceptual operation in its own class
 - Supports different operations in different applications that share the overall object structure.
- Result
 - The object structure rarely change
 - The operations over the structure change frequently

Drawbacks

Adding new concrete element classes is difficult:

- ❑ All visitors must change
- ❑ Here putting functionality in structure probably better

New Classes? New Ops?	Rare	Frequent
Rare	Either	Structure
Frequent	Visitor	Hard



Template Method Pattern

Acknowledgement: Some material in these slides is adopted from Head First Design patterns

Starbuzz Barista Training Manual

Starbuzz Coffee recipe

- Boil some water
- Brew coffee in boiling water
- Pour coffee in cup
- Add sugar and milk

Starbuzz Tea Recipe

- Boil some water
- Steep Tea in boiling water
- Pour tea in cup
- Add lemon

Coffee Code

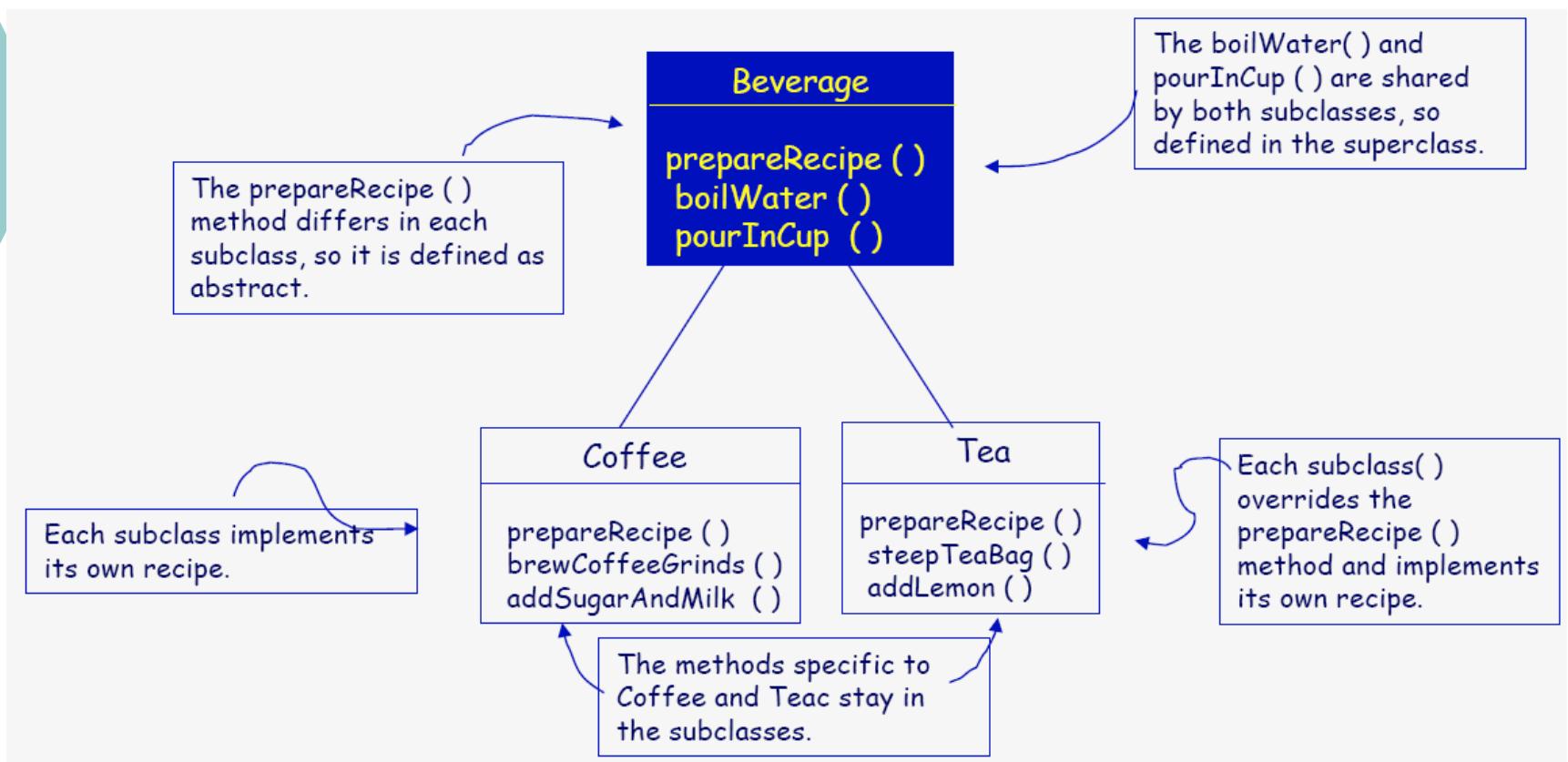
```
public class Coffee {  
    void prepareRecipe () {  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
  
    public void boilWater() {  
        System.out.println ("Boiling Water")  
    }  
    public void brewCoffeeGrinds() {  
        System.out.println ("Dripping coffee though filter")  
    }  
    public void pourInCup() {  
        System.out.println ("Pouring into Cup")  
    }  
    public void addSugarAndMilk() {  
        System.out.println ("Adding Sugar and Milk")  
    }  
}
```

Tea Code

```
public class Tea {  
    void prepareRecipe () {  
        boilWater();  
        steepTeaBag();  
        pourInCup();  
        addLemon();  
    }  
  
    public void boilWater() {  
        System.out.println ("Boiling Water")  
    }  
    public void steepTeaBag() {  
        System.out.println ("Steeping the Tea")  
    }  
    public void pourInCup() {  
        System.out.println ("Pouring into Cup")  
    }  
    public void addLemon() {  
        System.out.println ("Adding Lemon")  
    }  
}
```

Exercise

Redesign class diagram to remove redundancy



Did we do a good job on the redesign? Are we overlooking some other commonality? What are the other ways that **Coffee** and **Tea** are similar

Taking the design further

Coffee recipe

- Boil some water
- Brew coffee in boiling water
- Pour coffee in cup
- Add sugar and milk

Tea Recipe

- Boil some water
- Steep Tea in boiling water
- Pour tea in cup
- Add lemon

Both recipes follow same algorithm

- Boil some water
- Use hot water to extract Coffee or Tea
- Pour coffee in cup
- Add the appropriate condiments to the beverage

Taking the design further

```
void prepareRecipe () {  
    boilWater();  
    brewCoffeeGrinds();  
    pourInCup();  
    addSugarAndMilk();  
}
```

```
void prepareRecipe () {  
    boilWater();  
    steepTeaBag();  
    pourInCup();  
    addLemon();  
}
```

```
void prepareRecipe () {  
    boilWater();  
    brew();  
    pourInCup();  
    addCondiments();  
}
```

Beverage Code

```
public abstract class CaffeineBeverage {  
  
    void prepareRecipe () {  
        boilWater();  
        brew();  
        pourInCup();  
        addCondiments();  
    }  
  
    public void boilWater() {  
        System.out.println ("Boiling Water")  
    }  
  
    abstract void brew();  
  
    public void pourInCup() {  
        System.out.println ("Pouring into Cup")  
    }  
  
    abstract void addCondiments();  
}
```

Tea and Coffee Code

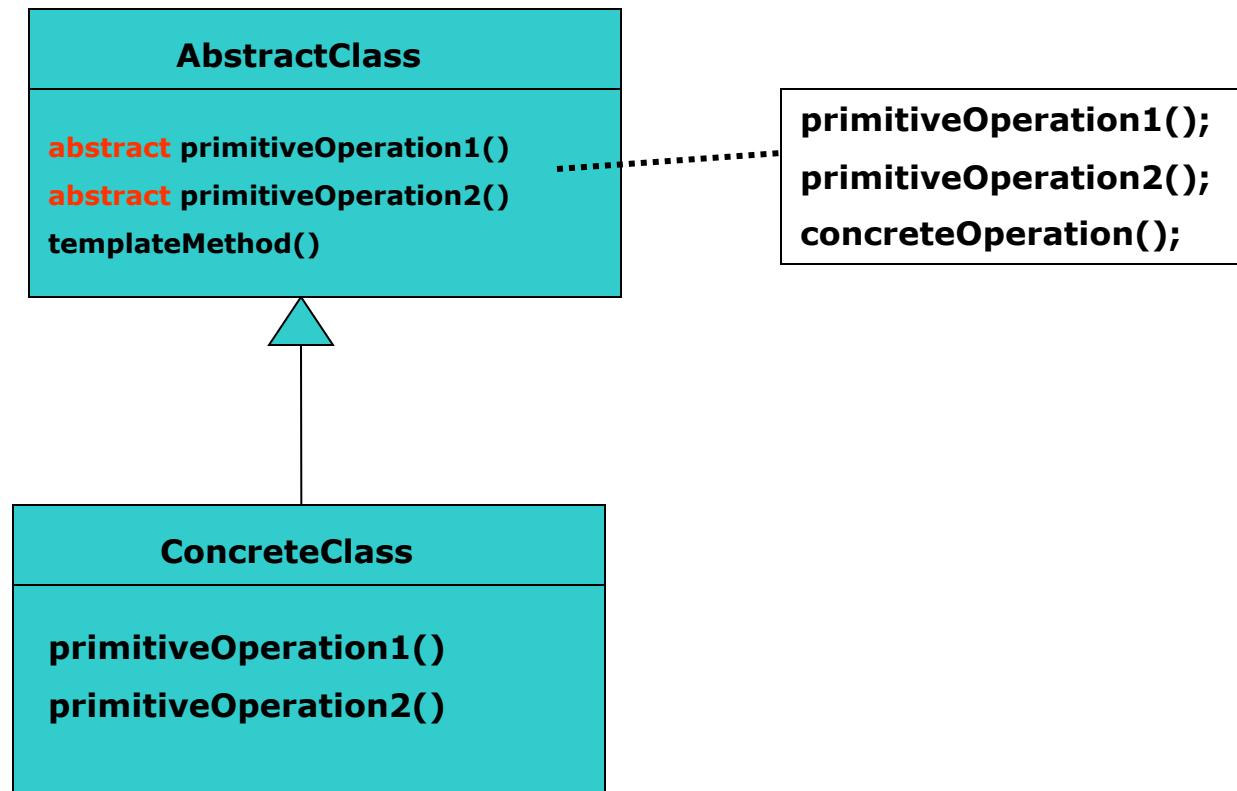
```
public class Coffee extends CaffeineBeverage {  
    public void brew() {  
        System.out.println ("Dripping coffee though filter")  
    }  
    public void addCondiments() {  
        System.out.println ("Adding Sugar and Milk")  
    }  
}
```

```
public class Tea extends CaffeineBeverage {  
    public void brew() {  
        System.out.println ("Steeping the tea")  
    }  
    public void addCondiments() {  
        System.out.println ("Adding Lemon")  
    }  
}
```

Template Method Pattern Defined

- The Template method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template method – class diagram



When to use...

- Let subclasses implement alternate behavior by method overriding
- Avoid code duplication
- Super class method still maintains control compared to a simple polymorphic override, where the base method is entirely rewritten



Hollywood Principle

Don't Call us, We'll Call you!

Important OO Principle - Composition over inheritance !

- Classes should achieve polymorphic behavior and code reuse by their composition rather than inheritance
 - Better Testability
 - Inheritance may break encapsulation
 - More flexibility (with replacement of composed class implementation) – for example, if you are using a Comparator class, changing to a different type is easier at run-time



The Strategy Pattern

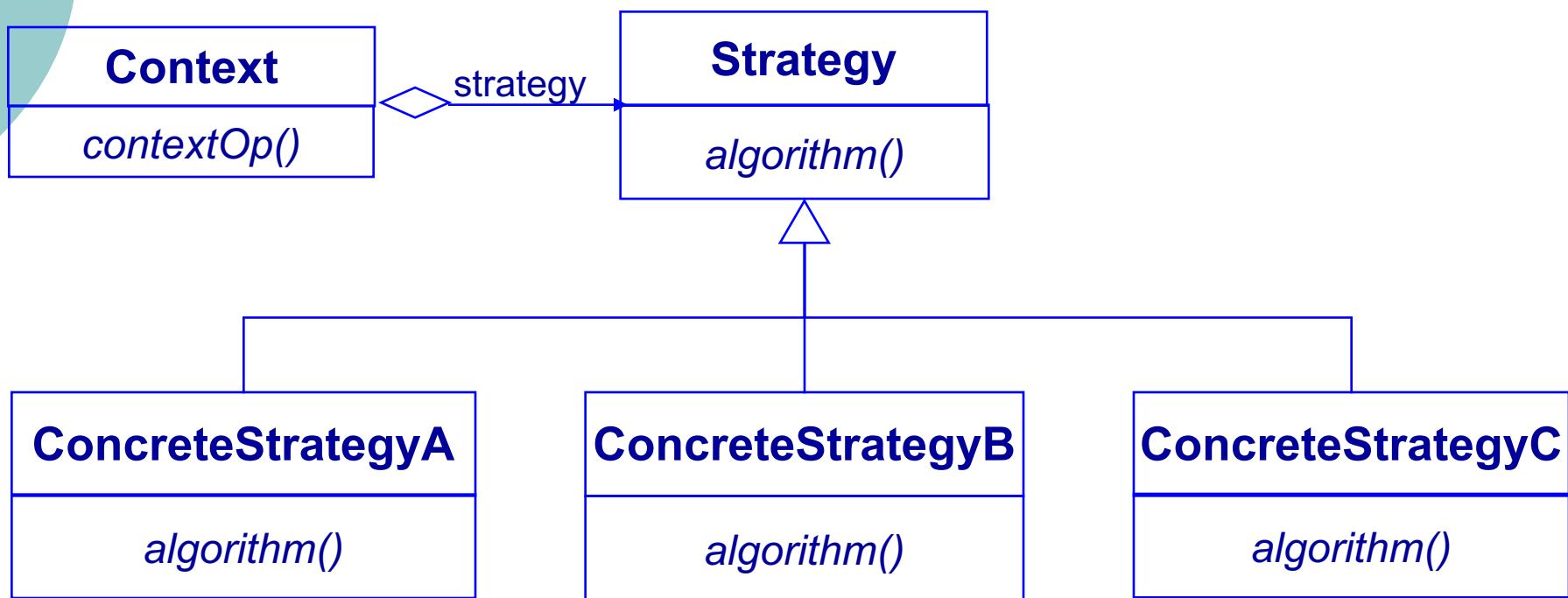
Motivation

- Problem
 - You have a family of related algorithms
 - Sorting and Searching
 - Line Breaking and Page Layout
 - AWT Layout Managers
- Desire
 - Encapsulate the algorithms so they can be used interchangeably
 - Isolate clients from the specific algorithm employed
- Solution
 - Define a common interface for each group of algorithms
 - Encapsulate algorithms in a *Strategy* (aka Objects as Algorithms)

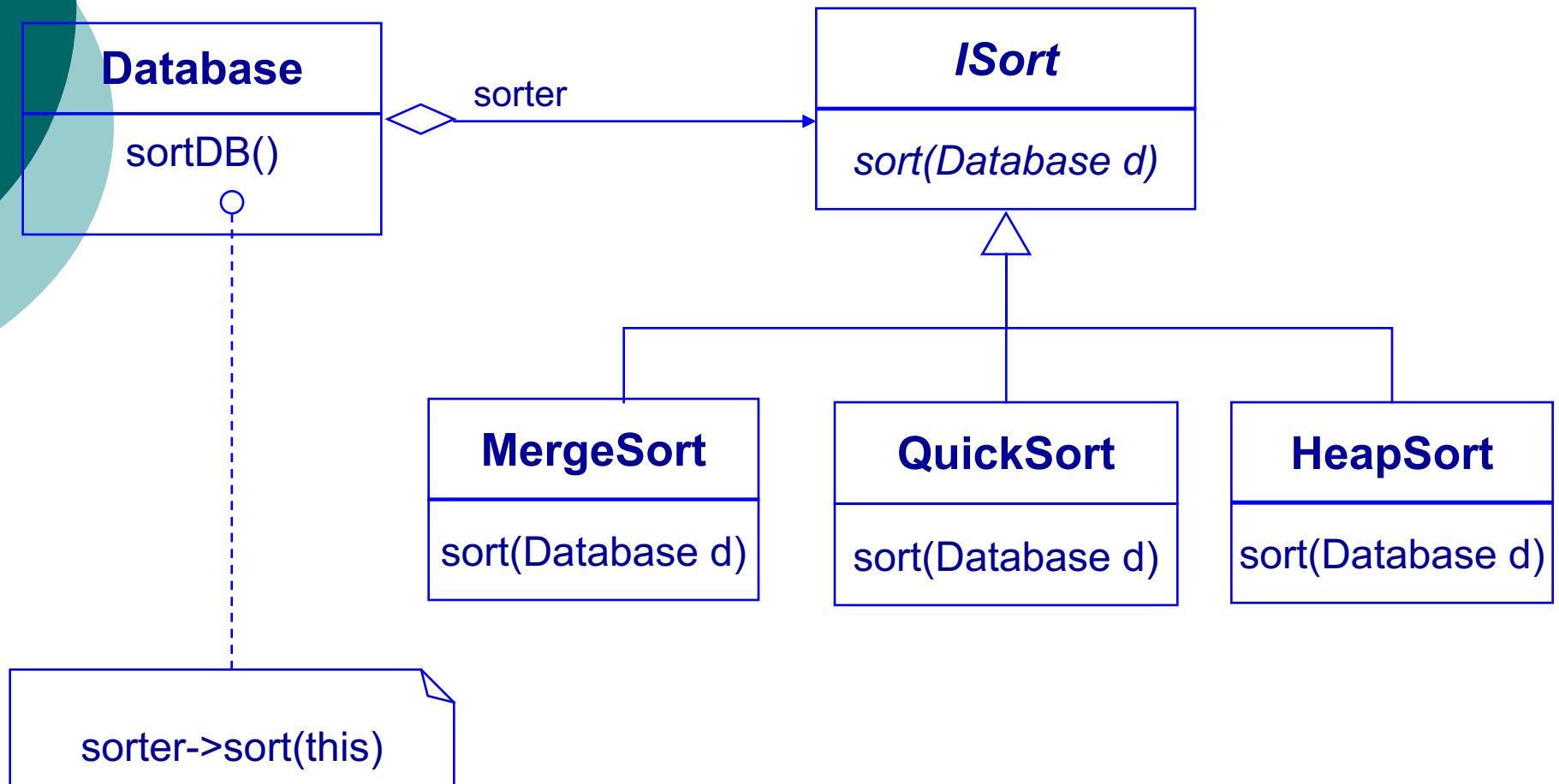
Strategy Pattern

- Provide invariant interface to varying algorithms:
 - Java interface
 - Java abstract class / C++ pure virtual functions
- Subclass for each algorithm variant
- Parameterize clients by interface

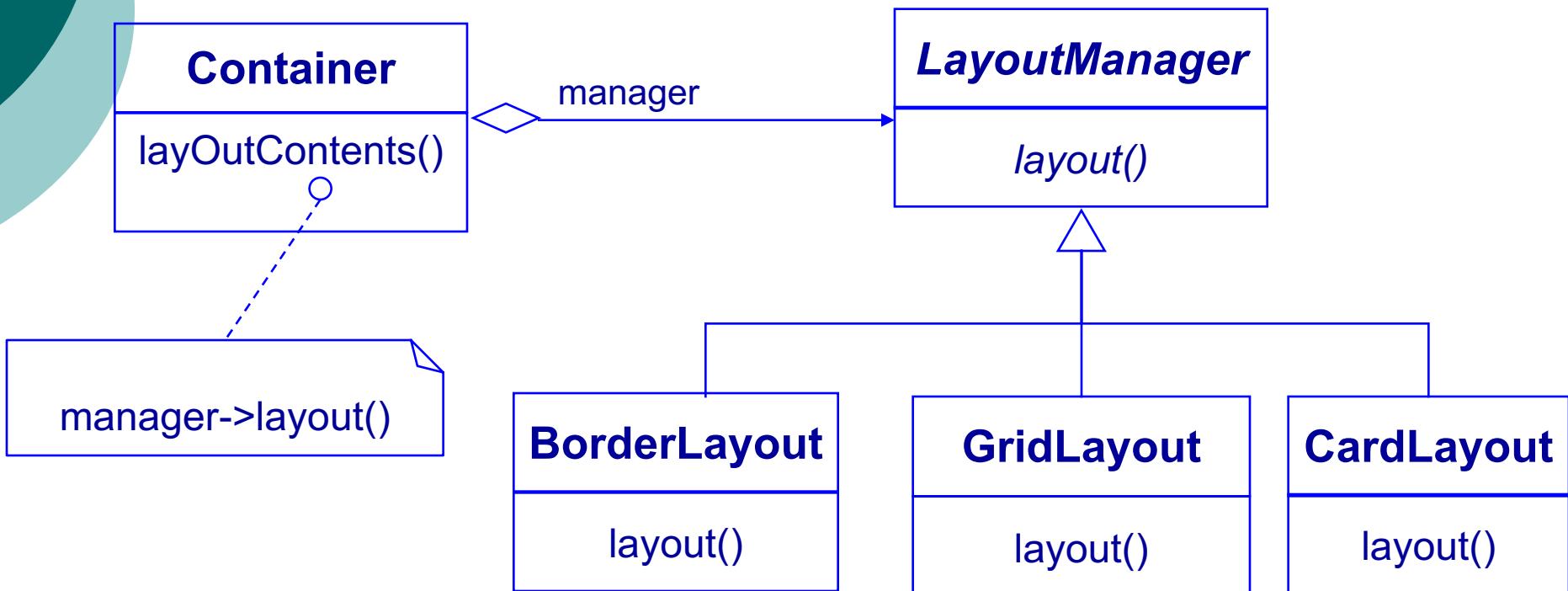
Strategy Pattern Structure



Example: Sorting a Database



Example: AWT Container Layout



Participants

- **Context**
 - Configured with a particular ConcreteStrategy to use
 - Keeps reference to (current) Strategy
 - May define interface for strategy to call back.
- **Strategy**
 - Common interface to supported algorithm(s)
 - Used by context to access the actual algorithm(s)
 - Abstract / virtual
- **Concrete Strategy**
 - Implements specific algorithm(s)

Applicability

- Many related classes differ only in specific reaction to requests.
- Need different variants of an algorithm.
 - Sorting?
 - Retrieval?
- Hide algorithm specific data structures
- To eliminate multiple conditional tests to select behavior in a client / context

Potential Benefits

- Encapsulates families of algorithms
- Alternative to subclassing *Context*
 - *Context* could be the top of a hierarchy.
 - Subclasses only change algorithm
 - Algorithm's implementation mingled with its use
 - Separate what varies from what is constant
- Eliminates often used conditionals and switches
- Context can choose alternatives
 - Choice is dynamic
 - Could choose different variants of the same algorithm

Potential Drawbacks

- Client must be aware of different strategies
 - At least some implementation issues exposed
 - Otherwise, how can client choose strategies?
- Communications overhead
 - Same interface for all Concrete Strategies whether simple or complex
 - Simple strategies may not use all provided information



The State Pattern

Motivation

- Problem:
 - An object responds differently based on its state
- Example: Fax transmitter
 - Basic operations:
 - boolean dial()
 - boolean getSpeed()
 - void send(Doc d)
 - Status hangUp()
 - Problems:
 - Not all operations legitimate at all times
 - Meaning of operation depends on current state

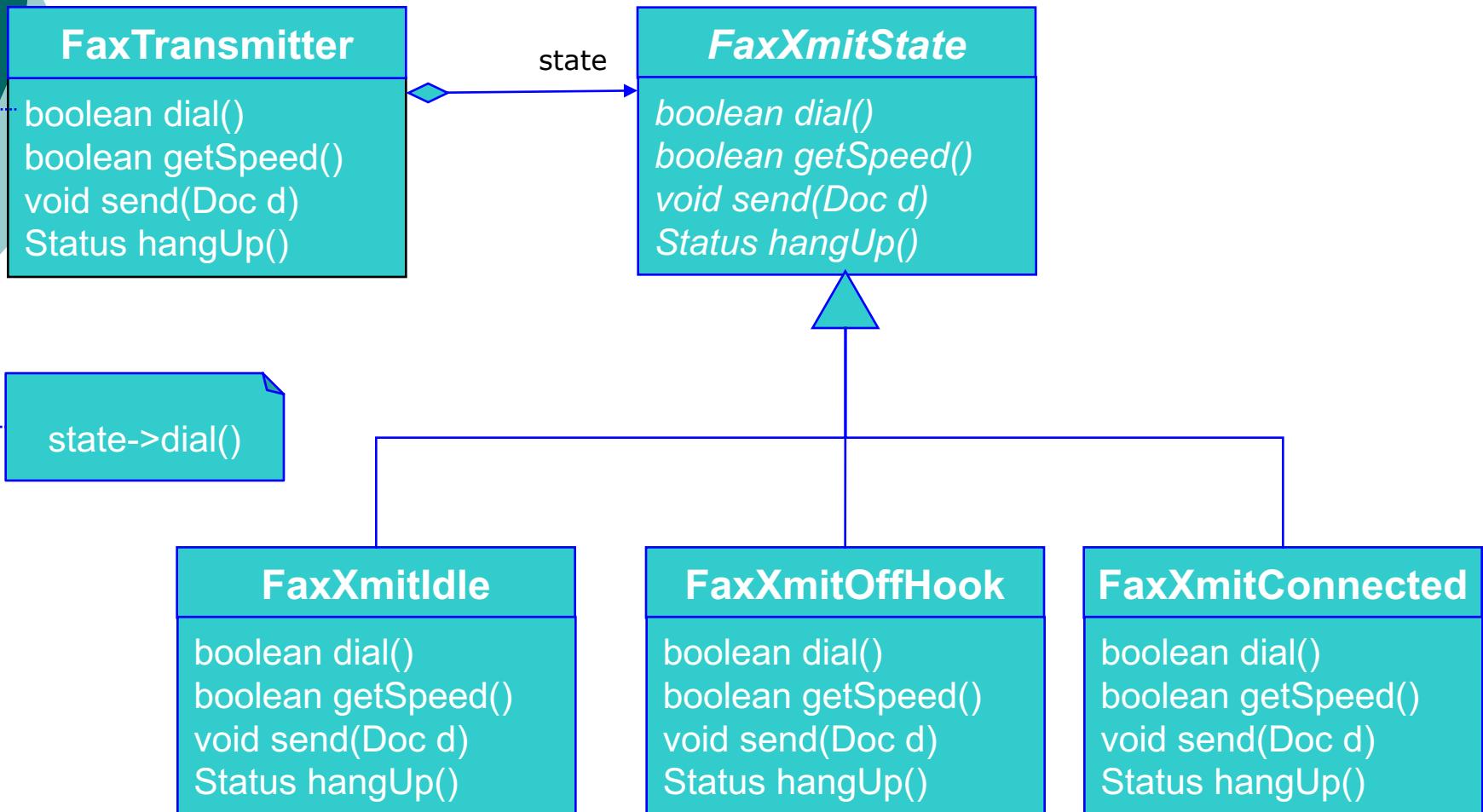
Naive Solution

- Create class FaxTransmitter
- Implement each operation directly
- Use variables to record state
- Use conditionals w/ state variables:
 - To decide whether request is legal
 - To decide on method of handling request
 - To decide whether to ignore request
- Issues
 - Complex, logic in methods
 - Hard to add new states (retry?)
 - Inflexible, hard-to-extend design

Better Solution

- Define basic interface abstractly
- Subclass for distinct response sets (state)
- Client's communicate with a *Context* object
- *Context* switches state object under the hood
- To clients, *Context* seems to change its class

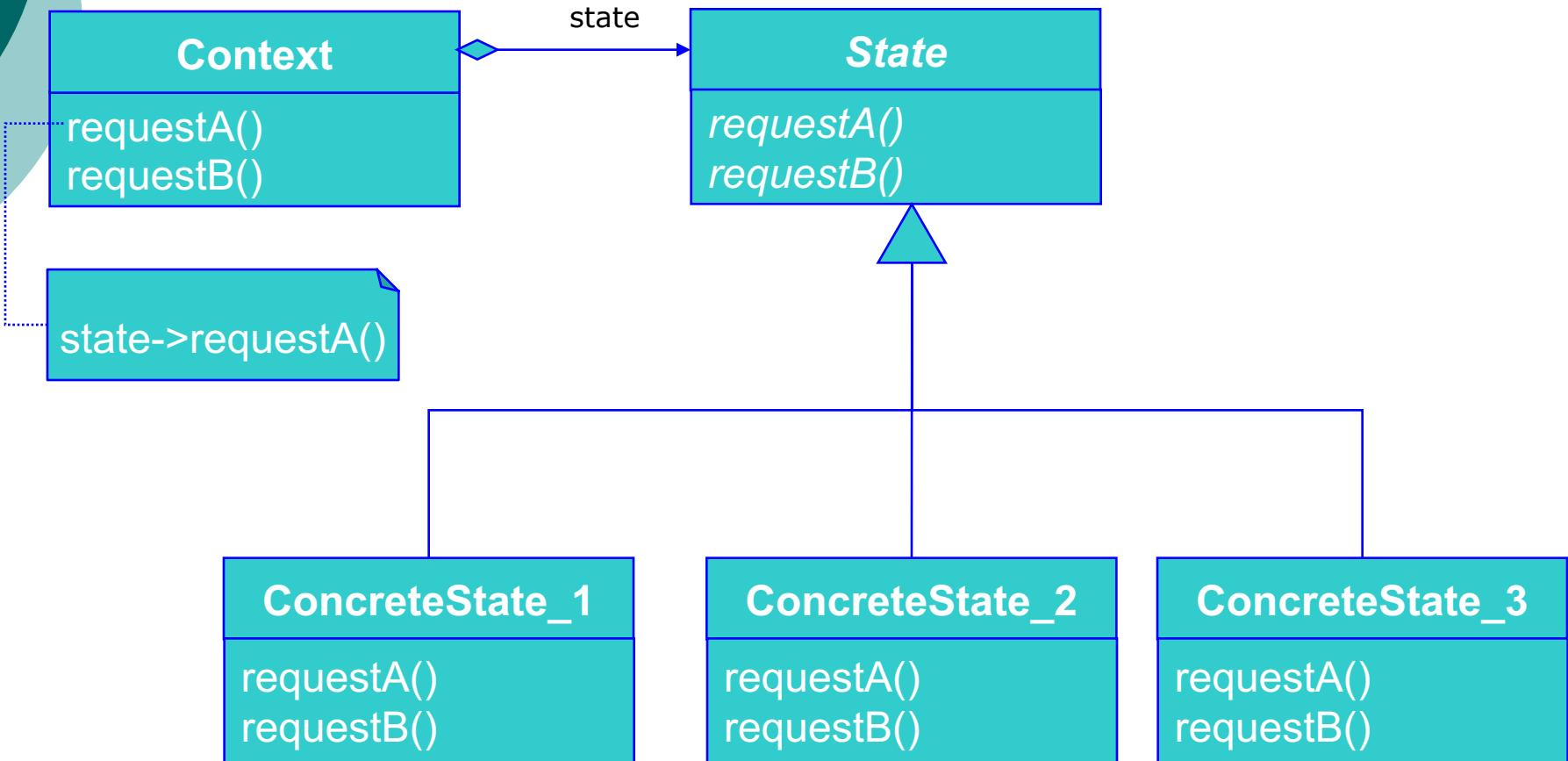
Example: FaxTransmitter



The State Pattern

- **Context** (`FaxTransmitter`) object defines interface
- Forwards request to a **delegate** (`FaxXmitState`) object representing current state
- Delegates have **same / similar interface** as Context
- Subclass Delegate for each concrete state
- State switch => delegate switch
- **Clients** see **Context** as having changed state

Pattern Structure



Participants

- Context (**FaxTransmitter**)
 - Defines interface of interest to clients
 - Keeps reference to current state
 - Forwards state-specific requests to current state object
- State (**FaxXmitState**)
 - Abstracts state specific behavior (methods)
- Concrete State (**FaxXmitIdle**)
 - Implements behavior for a specific state

Applicability

- Use when object's behavior depends on state that changes at run-time
- To factor out operations with multi-part conditionals based on object's state
 - State pattern puts each conditional in different concrete state class
 - Treats states as entities that can vary independent of context

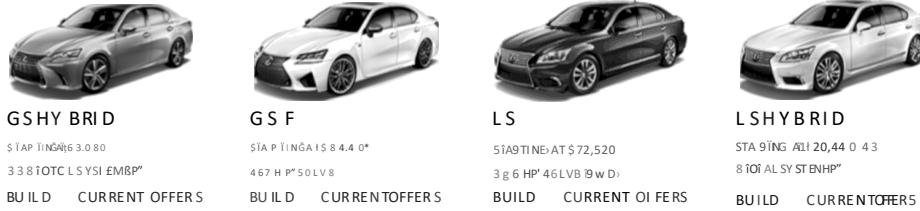
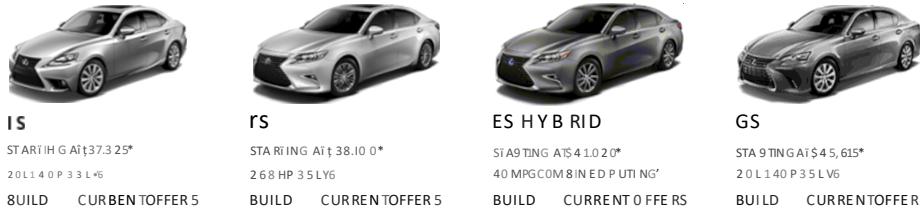
Consequences

Localizes & partitions state-specific behavior

- Replaces large, complex conditionals with many state objects
- Eases addition of new states and transitions

USABILITY

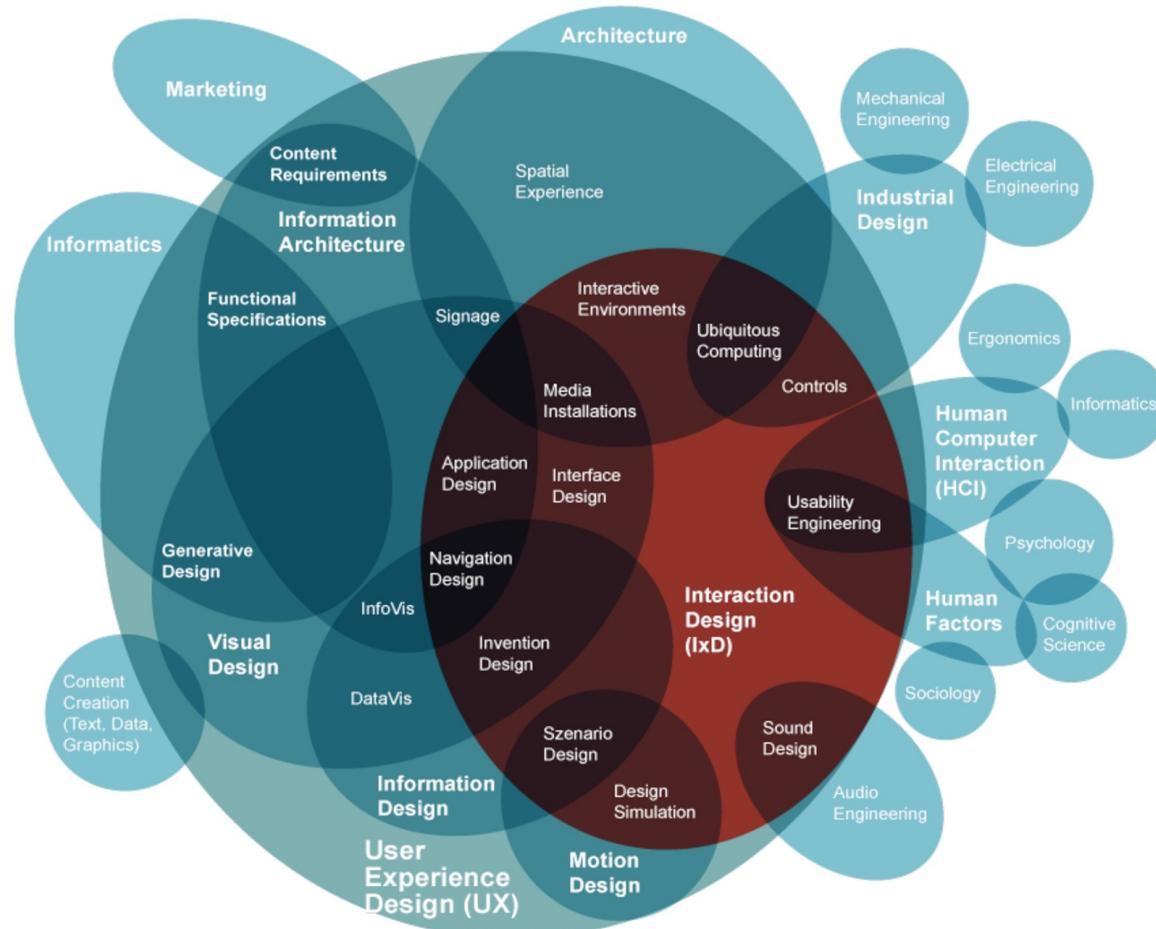
What happens outside (Customer?) and
What happened inside (Engineering?)



Different Design Aspects

- Architecture design
 - Division into subsystems and components
 - How these will be connected
 - How they will interact
 - Their interfaces
- Class Design
 - Various features of classes
- User Interface design
- Algorithm design
- Protocol design
 - Design of communication protocol

User Experience Design



Copyright :envis precisely (2009)
based on »The Disciplines of User Experience« by Dan Saffer (2008)
www.kickerstudio.com/blog/2008/12/the-disciplines-of-user-experience

UI Failures



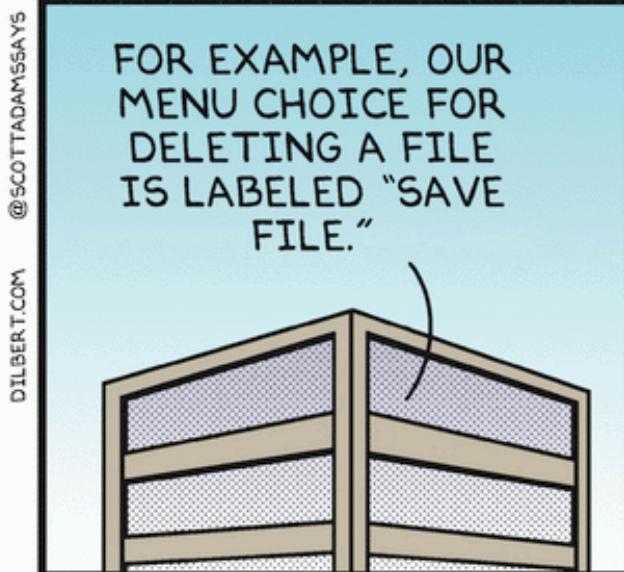
CUSTOMERS ARE
COMPLAINING BECAUSE
OUR USER INTERFACE
IS CONFUSING.



@SCOTTADAMSSAYS

DILBERT.COM

FOR EXAMPLE, OUR
MENU CHOICE FOR
DELETING A FILE
IS LABELED "SAVE
FILE."



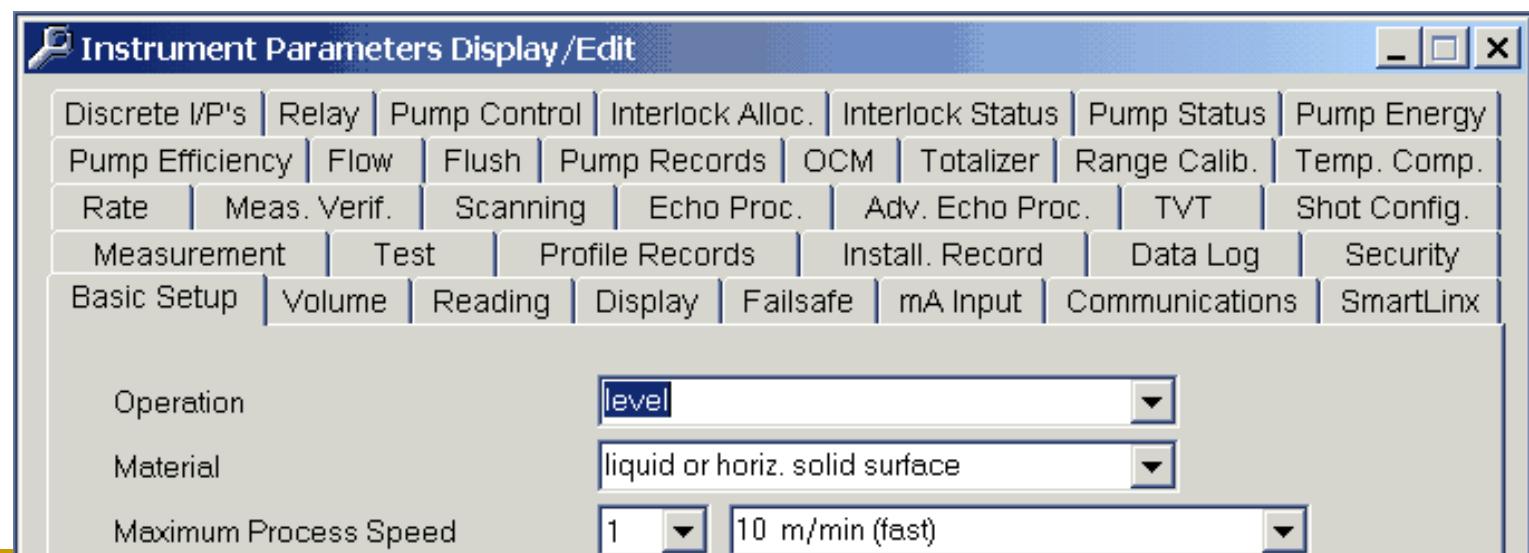
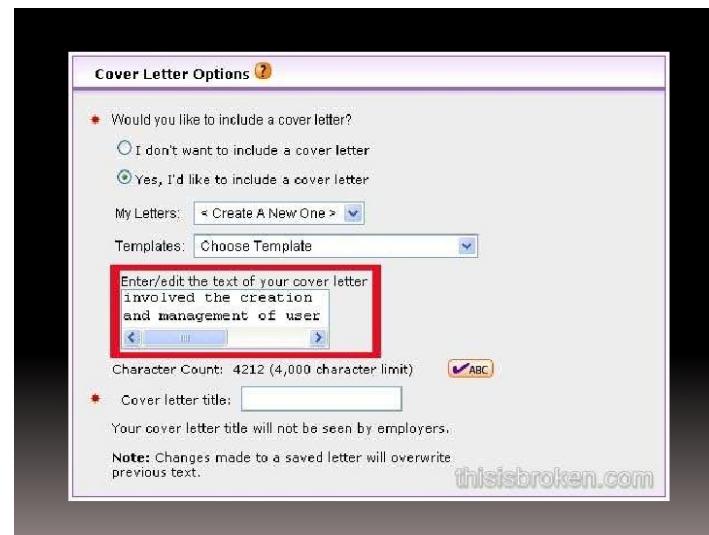
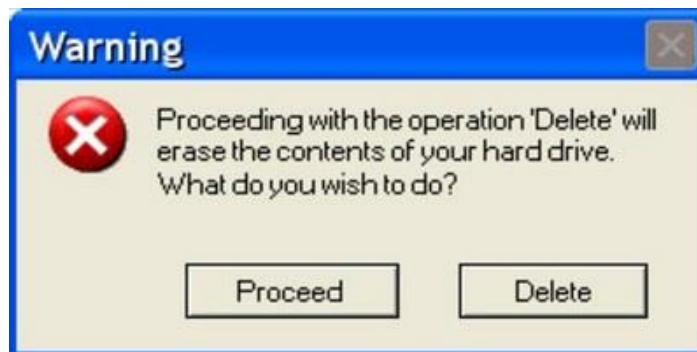
5-2-18 ©2018 Scott Adams, Inc./Dist. by Andrews McMeel

THAT'S
WHY WE
HAVE A
HELP
MENU.

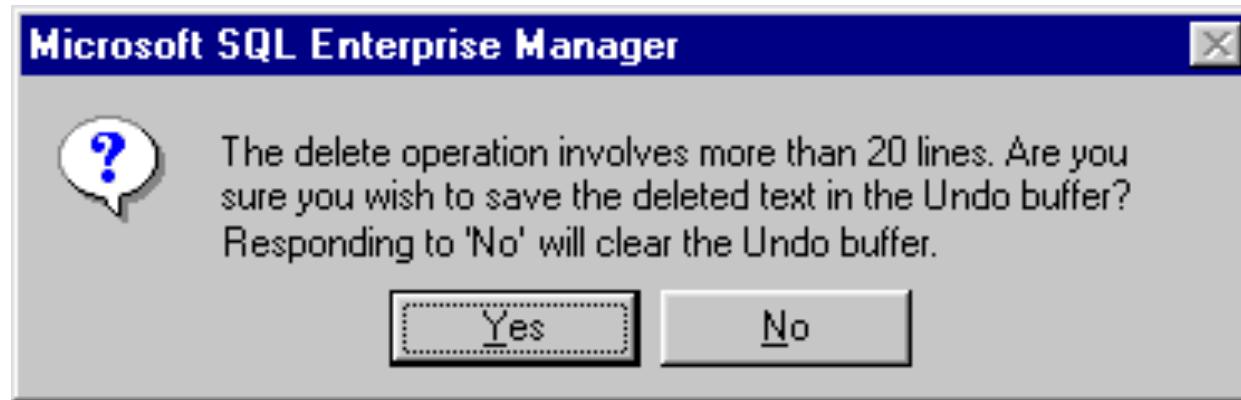


OUR HELP
MENU IS
LABELED
"REFORMAT
HARD
DRIVE."

UI failures contd...



UI failures contd...



8) Age:

9) Female
 Male

Enter your Social Security Number:

 - -
0 0 0 - 0 0 - 0 0 0 0

***** AT&T M-Cell ⌂ 10:16 AM

espn.go.com

CORPORATE USA EXPORTED More ▾ CITY: BOSTON CHICAGO DALLAS LOS ANGELES NEW YORK Signal Register

ESPN

NFL NFL Playoffs Final PBL NASCAR Stock NCAP GOLF (M) NCA All Scores ▾ Regular Season Week 8

my ESPN NFL MLD NCA NHL NCAAF NCAAM SOCCER MORE SPORTS WATCH FANTASY espnRADIO

THIS IS YOUR GAME

TOPICS Playoff Committee Factors W2 Game 8 Square Defense

Evaluating Romeo's injury

This Video is not authorized for playback on this device

King's Reign Continues

Expecting someone else? LeBron James takes the NBA's best talents back to Cleveland. #NBABucks vens. #Pacers MVP awards? #FiveThirtyEight: Coal's best - Week's best Team rankings: Steelers - Lions -

Additional: Rd. 1 Playoff Preview Skies High Up? TLA World Series Who Took Best?

BILL SHAWNS PRESENTS GRANTLAND

Tiers of the NBA Ranking from top to bottom: League + Upon Close Examination No surprises in Chapel Hill: Pearce + Cozen Say, Week 9 Res. B.S. Report + JACKIE MACBRIAN

Schilling Not Hiding Scars Cut Schilling opens up about his growing battle with cancer and his beloved company. Columns + FIVE THIRTY EIGHT

NBA Preview: This Warriors are best in the West. Columns + World Series What Royals may come to regret. Columns + ESPN FANTASY GAMES

your win streak ? Order

NBA Fantasy NBA Fantasy Points

Fantasy Basketball

ESPN Fantasy Basketball

The NBA season is already here. Who are you drafting? Get Started + Fantasy Basketball

All Leaderboards +

VIDEO: HAVE YOU SEEN?

OTL: Will Committee Get It Right? [36]

Game 6 Preview [24]

Wild NBA Offseason [5-4]

PAGE 1 OF 3

COLLEGE NASCAR ESPN-NATION espn X GAMES

< >

***** AT&T M-Cell ⌂ 10:15 AM

espn.go.com

Pros Called Him "Straight Arrow" Learn his Swing Secrets Right Here! SquareToSquareMethod.com

ESPN myESPN SPORTS ▾ ▾

#NBA RANK

King's Reign Continues

Expecting someone else? LeBron James takes the NBA's best talents back to Cleveland.

Predictions: Preseason awards

MORE

◀ PREV 1/5 NEXT ▶

NFL

< >

www.united.com/web/en-us/default.cfm

Flight Status & Information
Timetable
Baggage Information
Traveling with Animals
Special Travel Needs
Airport Information
Inflight Services
Route Maps
Destination Information
Mobile Tools

Anytime

Anytime

Offer Code (optional):

Find a

www.united.com/web/en-us/content/

UNITED | About Us | News Center | Skin My Account | Contact Us | Help | Site Feedback

Home > Travel information > Mobile tools

Mobile tools

Your go-to app when you're on the go

United app now available for Apple® iPhone®, iPad® and iPod touch® devices and for Android, Windows Phone 8 and BlackBerry® 10 devices.

Our app is your travel toolkit. You can book United flights, check flight and upgrade statuses, view your MileagePlus account information, check in and receive a mobile boarding pass and more, directly from your mobile device. Available for iPhone®, iPad® and iPod touch® devices (as an iPhone app) and for Android, Windows Phone 8 and BlackBerry 10 devices, the United app offers convenient access to your travel information.

Key features include:

- Flight booking** – Book flights on United, including award travel, directly from the app on your mobile device
- Check-in and mobile boarding pass** – Store your mobile boarding pass for easy access at security and during boarding
- Passport scanning (iPhone and Android)** – Use your phone to scan your passport for international check-in
- Flight status** – Check your upgrade status and view inflight amenities, departure gates and more
- Flight reminders** – Receive automatic updates on your mobile device for selected flights
- Seat maps** – Select seats and choose Economy Plus® seating

Usability

- Broadly the above examples can be looked at from
 - Functional Perspective
 - User Perspective

Usability

ISO (ISO/IEC 9126-1:2001) defines usability as

“The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use”

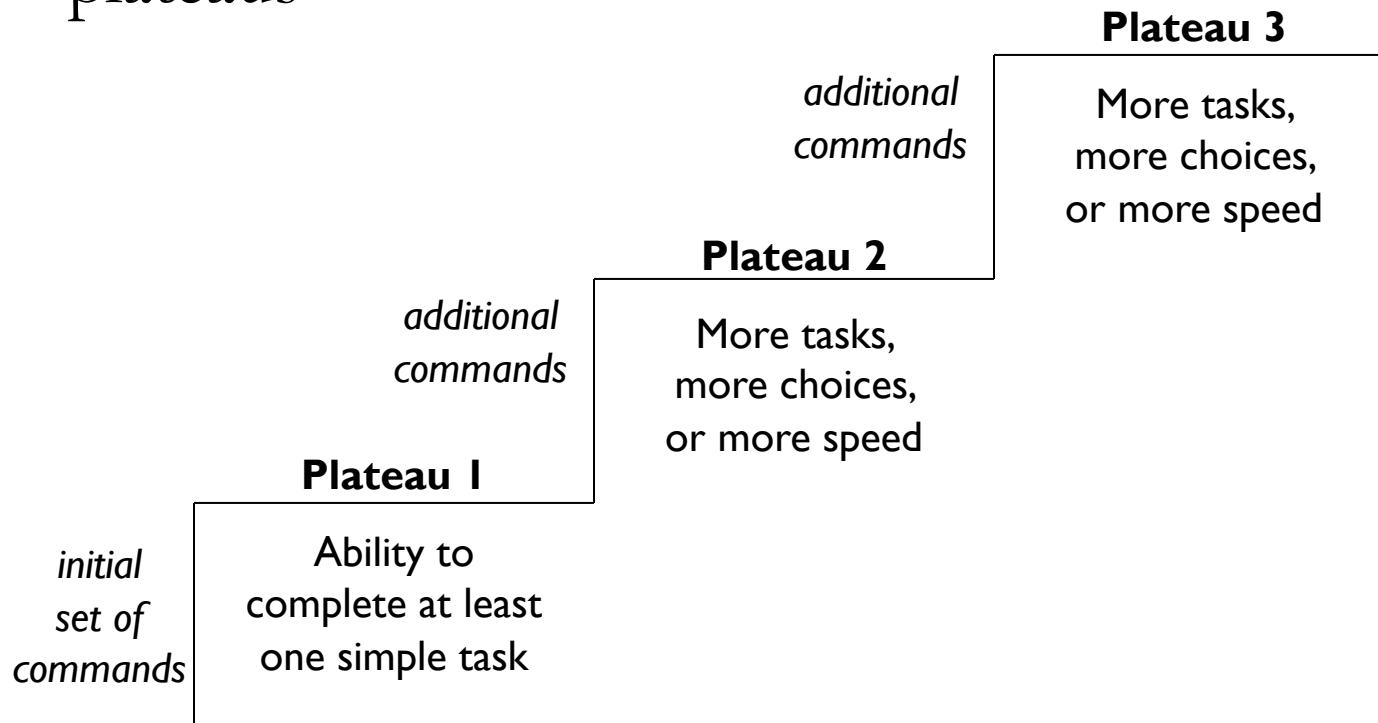
Attributes for Evaluating Usability

- i. Learnability
- ii. Efficiency
- iii. Memorability
- iv. Errors
- V. Satisfaction



1. Time to Learn (Learnability)

- How long it takes to learn how to use an interface
- With complicated interfaces, learning happens in “plateaus”



2. Speed of Performance (Efficiency)

- Speed of user interface, **NOT** software
- Number of characters to type, buttons to press, mouse-clicks, mouse movements, ...
- Speed of performance often directly conflicts with time to learn
 - That is, faster systems are often harder to learn
 - Command lines vs. GUIs

3. Rate of User Errors

- A UI needs to be designed in such a way that user mistakes are less likely
- Affected by factors such as :
 - Consistency
 - Instructions
 - Logical arrangement of screens
- Importance depends on criticality of software

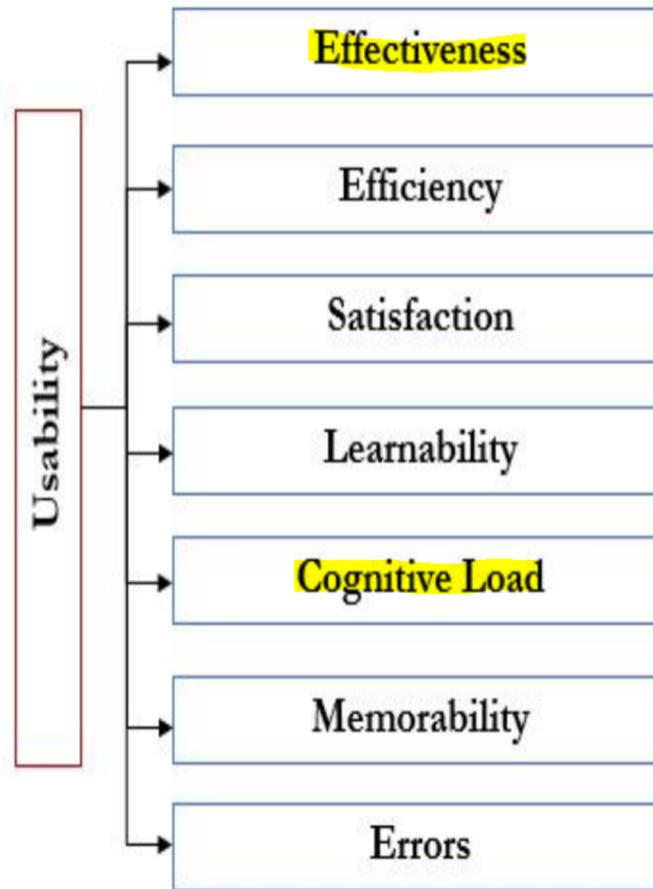
4. Retention of Skills (Memorability)

- We quickly forget how to use some user interfaces, but remember others for life
 - Calculus vs. algebra
 - Airplanes vs. bicycles
- Affected by how closely the syntax of the operations match our understanding
- If learning is very fast, retention may be less important

5. Subjective Satisfaction

- How comfortable the users are with the software
- Previous criteria are very analytical, objective, and measurable
- Subjective satisfaction captures other issues that are more specific to individual taste and background
 - A little harder to measure

People At the Centre of Mobile Application Development (PACMAD)



Design of User Interfaces

- Inside-out design :

Develop a system, then add the interface

- Outside-in design :

Design the interface, then build the system to support it

When decisions are made, either the developer must conform to the user, or the user must conform to the developer.

Traditional CS is entirely inside out

11 Reasons Mobile Apps Fail

The mobile industry is rapidly growing to an endless scope. But increased competition also leads to increased chances of failure.



by Lauren Gilmore · MVB · Jul. 05, 18 · Mobile Zone · Opinion

User Issues

There's nothing more frustrating than a mobile app that constantly crashes or has poor performance. Poor performance - or no performance - is the number one reason why users abandon an app.

6. Bad User Experience

Offering a compelling user experience is crucial for the success of any mobile app. While having a wide range of available features are key, user interface design is a major factor in determining whether an app is user friendly or difficult to use and causes performance issues.

Solution: Designate enough attention to UX/UI best practices and make sure your app is easy to use.

testbytes

Making Quality a Habit

Top 8 Reasons for App Failure and How to Avoid Them

Tuesday August 7, 2018

App Testing

5. Poor User Interface

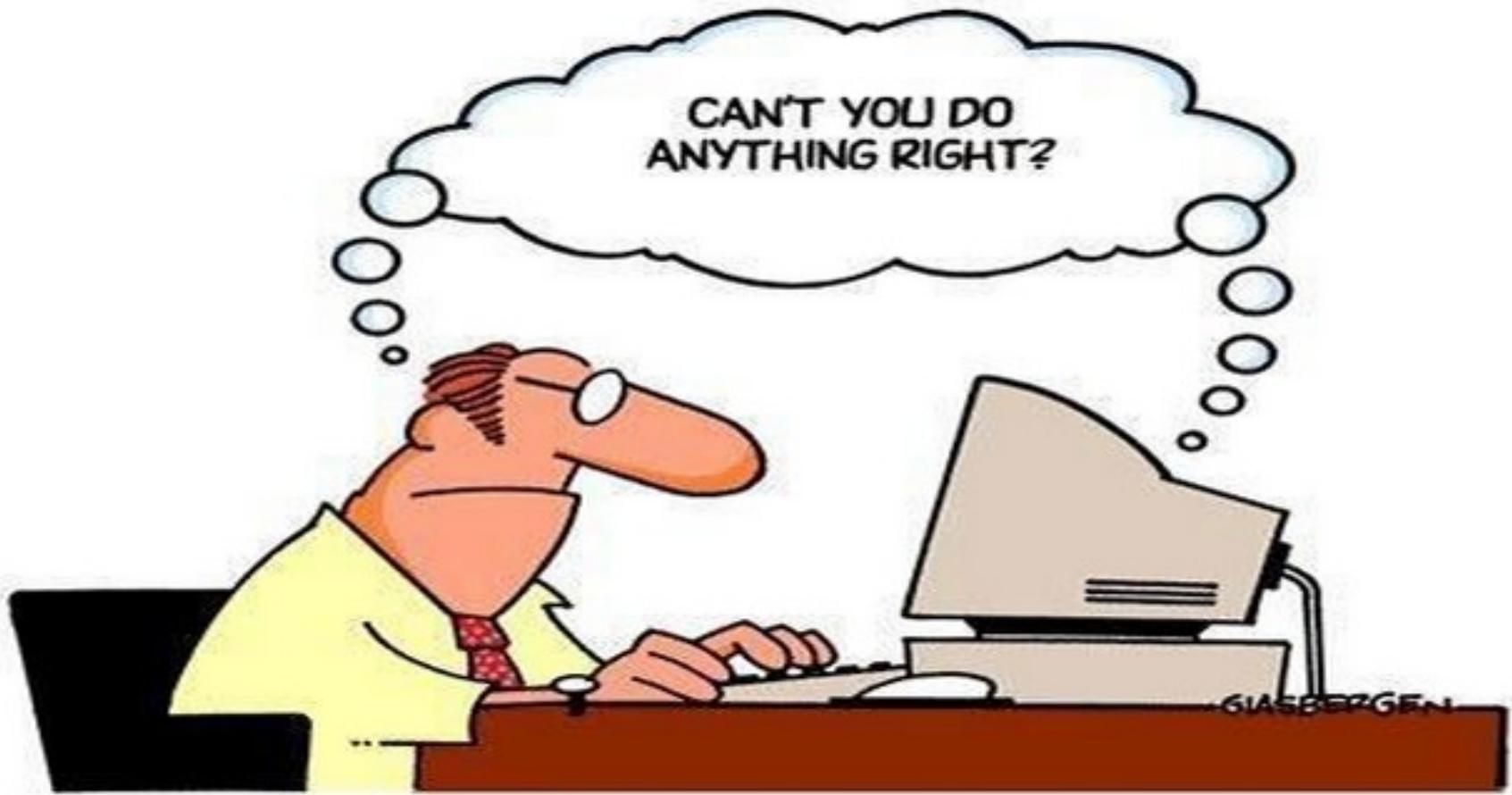
Generally, poor user interface means a poor design. Today, most apps are rejected by users due to their poor user interface. This ultimately leads to their failure due to poor user experience. If the performance of an app is unsatisfactory to the users, they will uninstall it in no time. If users are unable to perform basic functions seamlessly, then it will be very hard for developers to sell their product. There are various reasons how an app can give poor user interface such as:

Why is Designing UIs Hard ?

- Good UIs take time to design
 - Many programmers are lazy
- Designing a good UI requires thinking like the user instead of an engineer
 - Engineers often think they are users
- Different users want different things

Engineers love features

They want to do everything the technology allows!



Why is Designing UIs Hard ?

- Designers are usually experts
 - They view all functions as having equal weight
- Marketing want UIs designed for beginners
 - They sell to beginners
 - Many have only novice semantic knowledge
- Most users are in between—**intermediates !**

Don't weld on training wheels

Understand the Users

- Work experience
- Computer experience
- Age
- Education
- Reading skills
- Language skills
- Work environment
- Task frequency
- ... many more possibilities

**It is important to
know who the user is**

Distribution of Users

Beginners

What does the application do?

What doesn't it do?

How do I print?

Where do I start?

Intermediates

Forgot how

Where is it?

Remind me ...

Can I undo?

More features?

Experts

Automate?

Shortcuts?

Can I change?

Customize?

Keyboard shortcut?

Dangerous?

UI Patterns

<http://ui-patterns.com/patterns>

Shneiderman's Golden Usability Principles

1. Strive for consistency
2. Design usable and discoverable shortcuts
3. Provide appropriate feedback
4. Yield closure
5. Provide appropriate error handling
6. Allow users to reverse (undo) all actions
7. Support internal locus of control
8. Reduce the Short Term Memory (STM) load

Principle (1/8) - Consistency



MacOS menu bar
remains consistent!
1980s to 2019 !

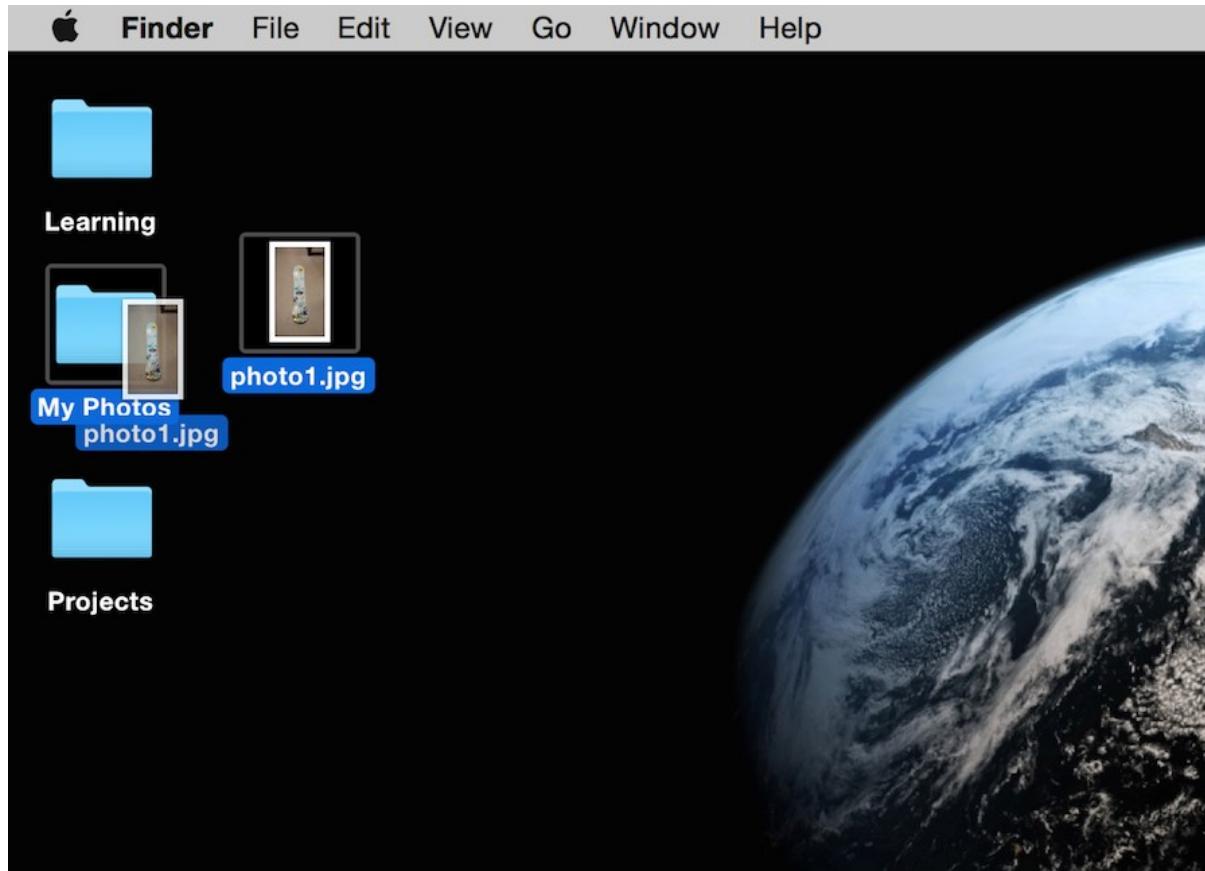
Various aspects of UI Must be consistent ! Changes break Consistency!!

Principle (2/8) – Shortcuts

Design usable and discoverable **shortcuts**

- ❑ Users must be able to find them
 - ❑ Users must be able to remember them
-
- Copy and Paste (Command-C and Command-V)
 - Screenshots (Command-Shift-3)

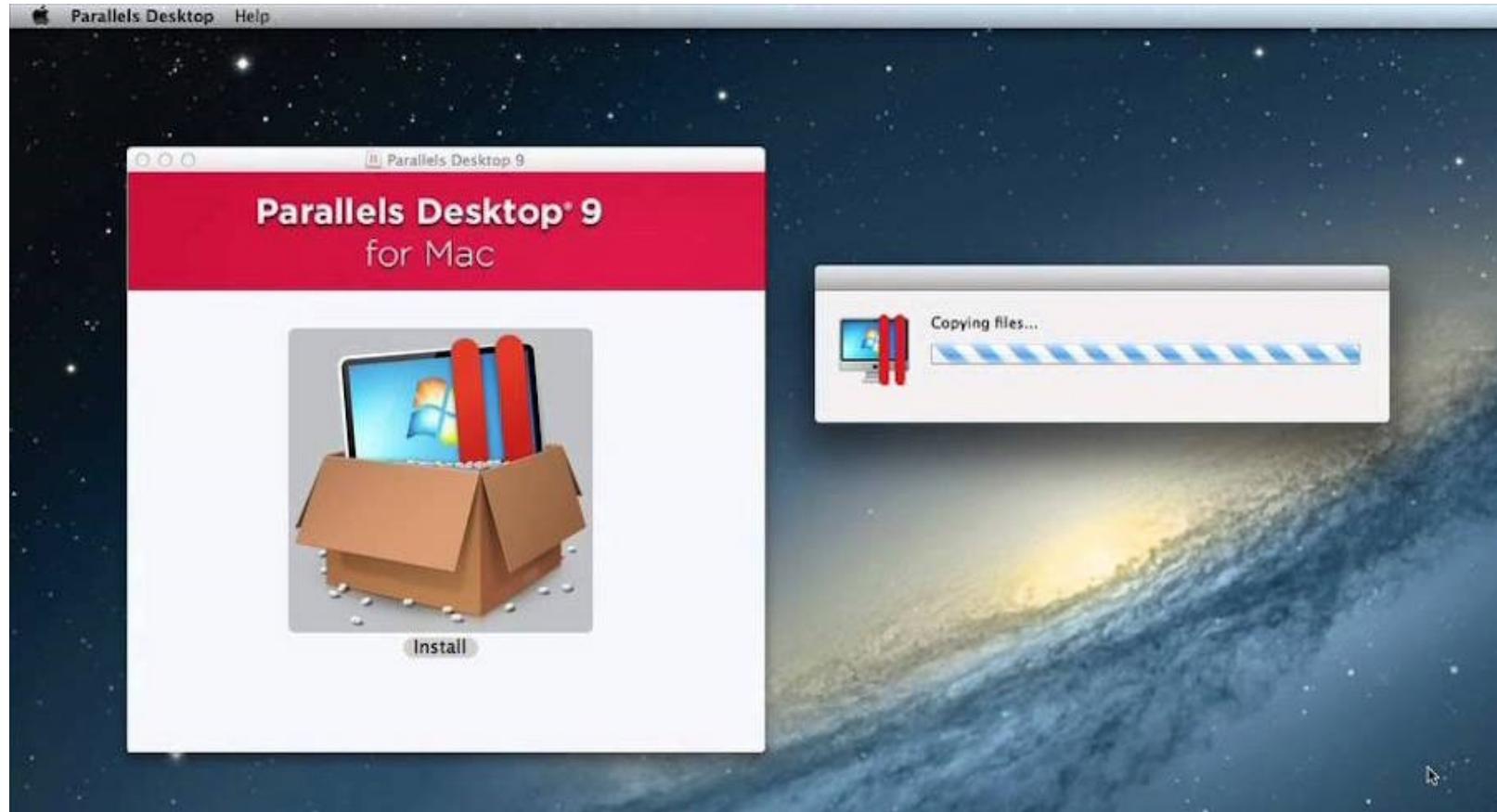
Principle (3/8) - Feedback



Folder is shown to be physically moving during drag and drop

Have a clearly defined end-point in the interaction !

Principle (4/8) – Yield closure



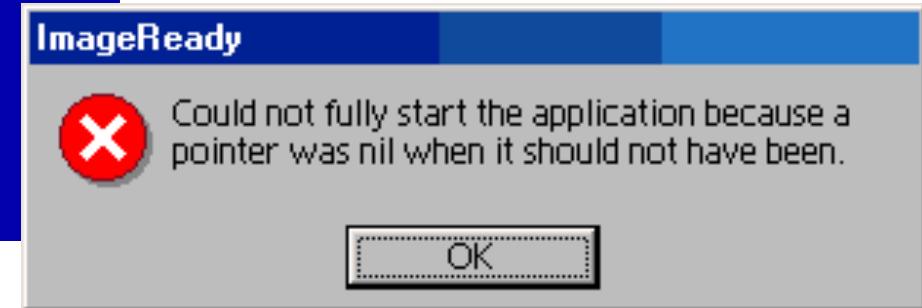
Appropriate feedback must be given (For errors as well as others) !

Principle (5/8) – Error handling

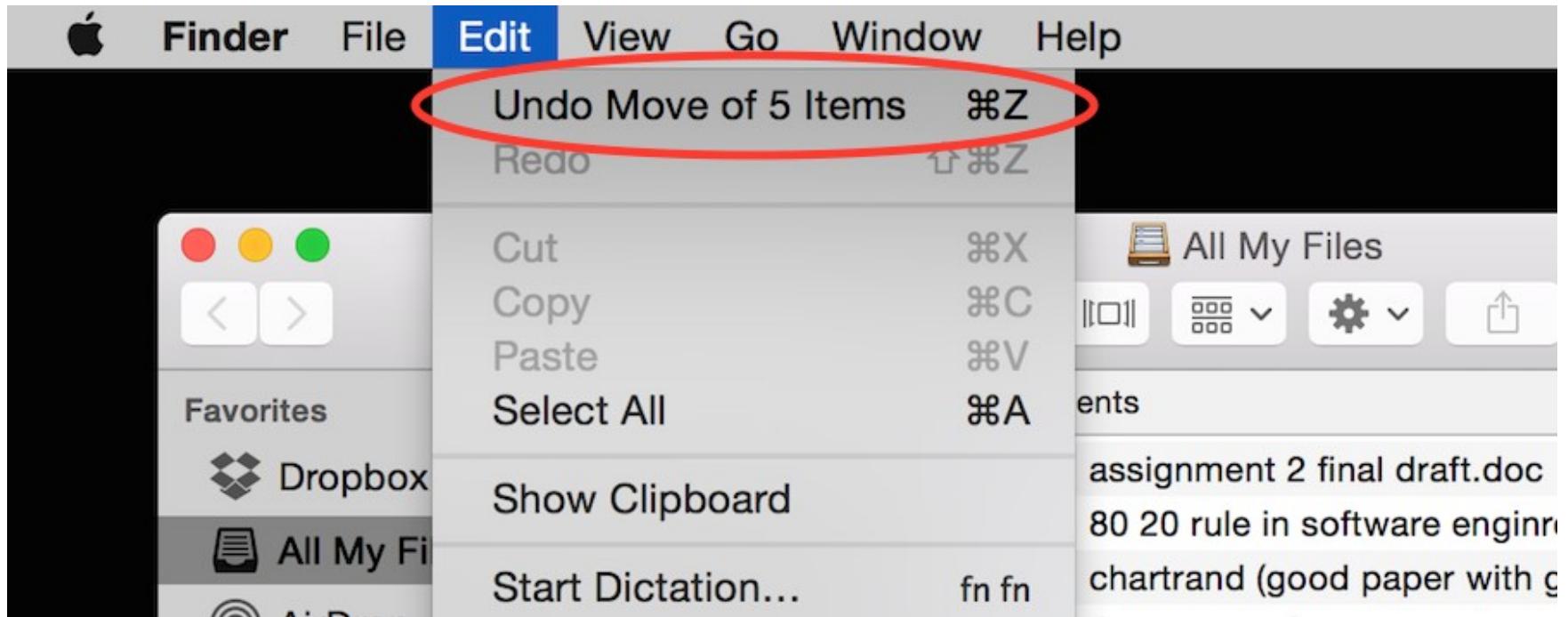


- Clearly tell users what was wrong
- Don't scare the users. Use appropriate language
- Don't just give Error handles or Error codes
 - “404 error” does not really convey much
- Only make users redo the part that was wrong

Principle (5/8) – Error handling

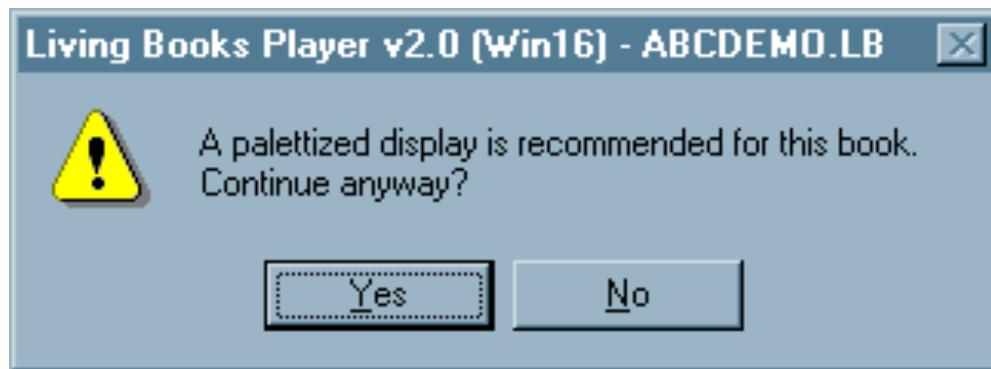
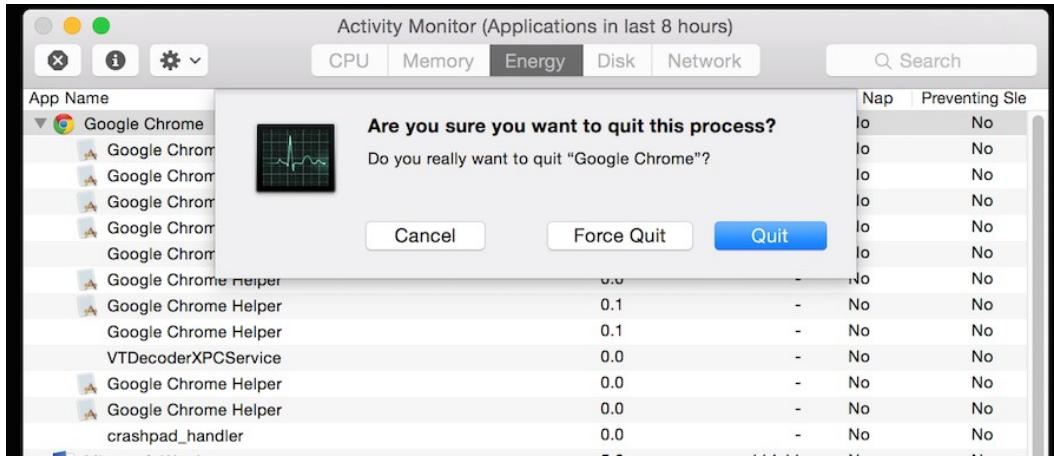


Principle (6/8) - UNDO



Actions must be reversible ! If some actions can't be undone, use “hesitation”

Principle (7/8) – Internal locus of control



- Inexperienced users may be intimidated when the software makes decisions
- Experienced users want to control the flow
- Put the users in-charge (to the extent possible)
 - Hide technical terms
 - Interaction must be customizable (Personalization)

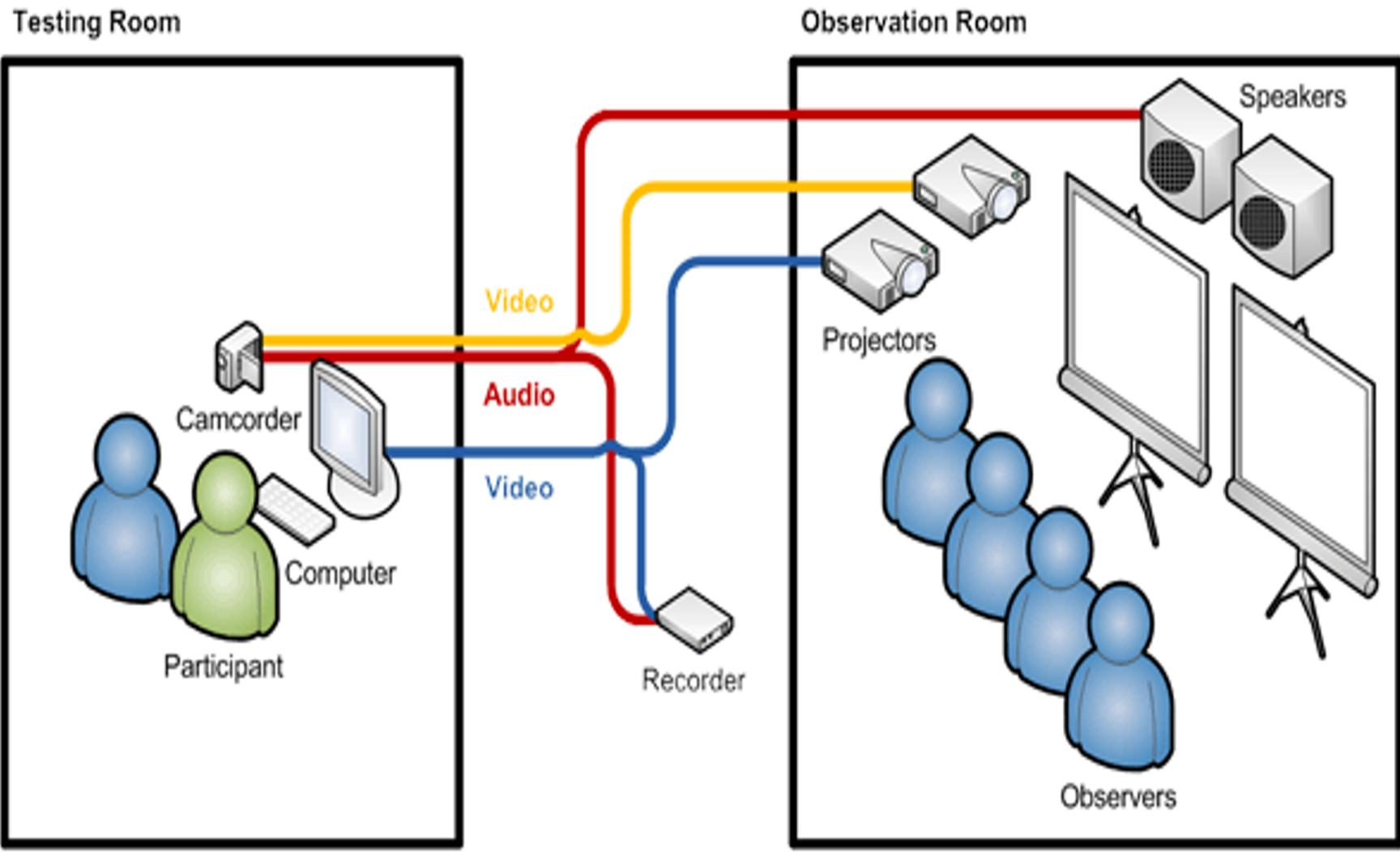
Principle (8/8) – Short term Memory (STM)

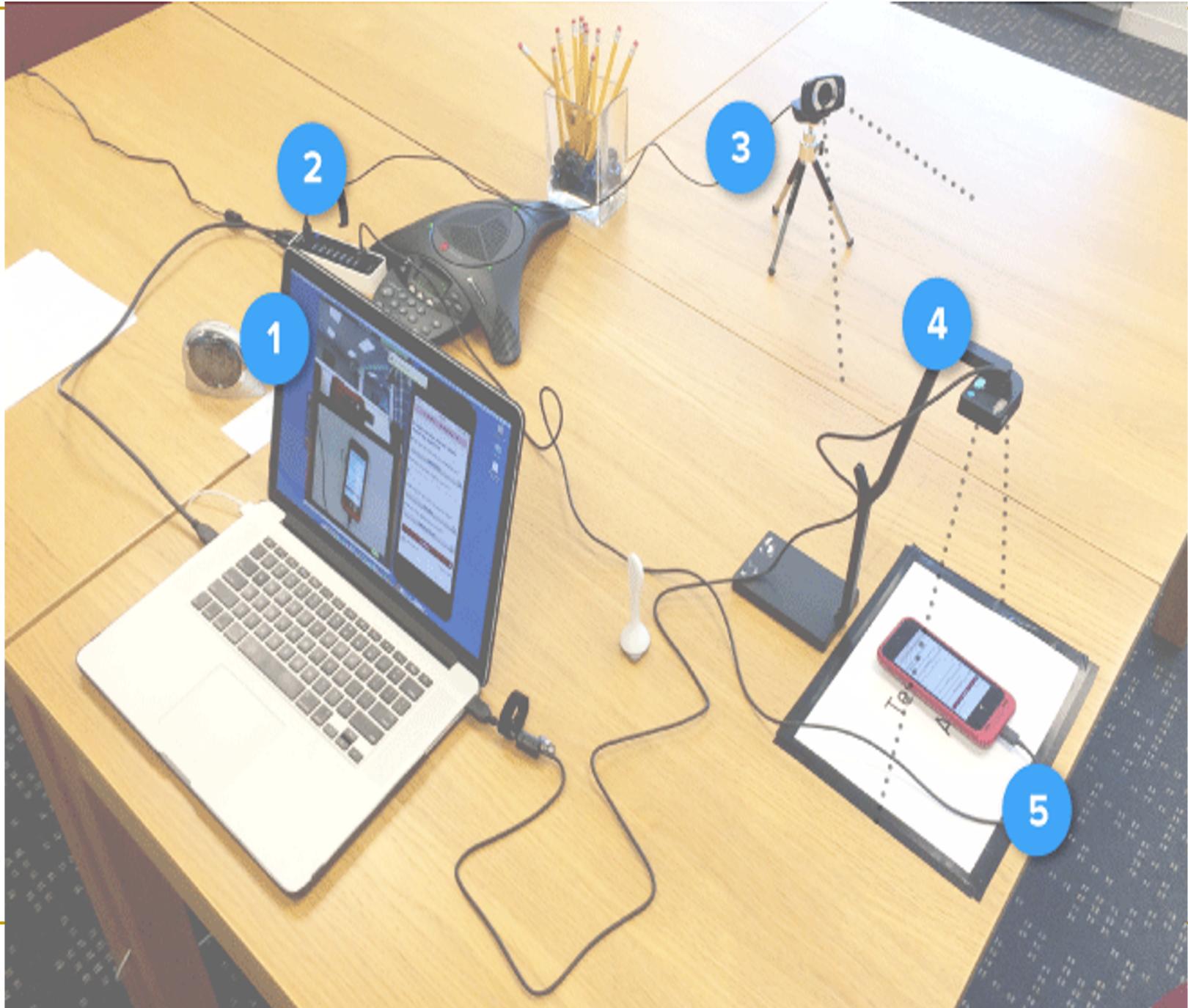


- Remember $7 +/ - 2$
- Apple has only 4 apps in the main menu
- Organizing things hierarchically help reduce STM

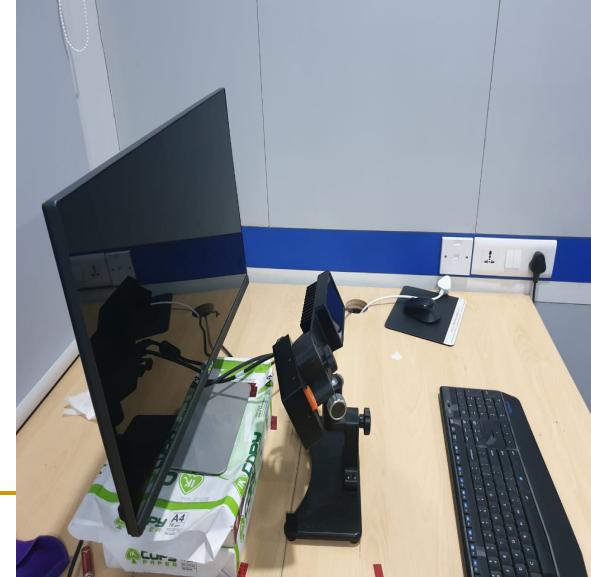
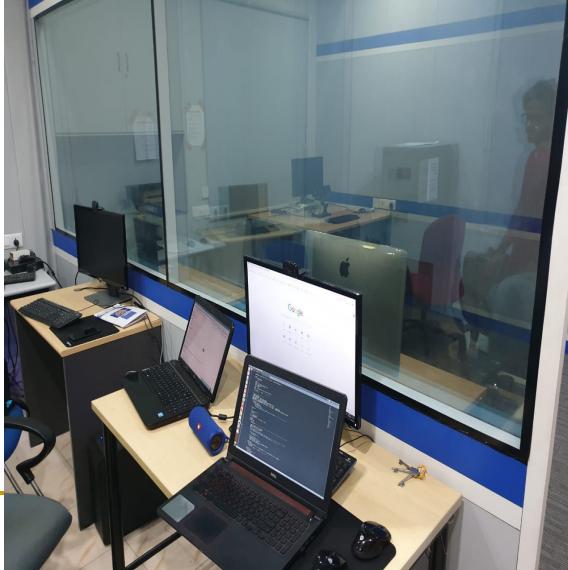
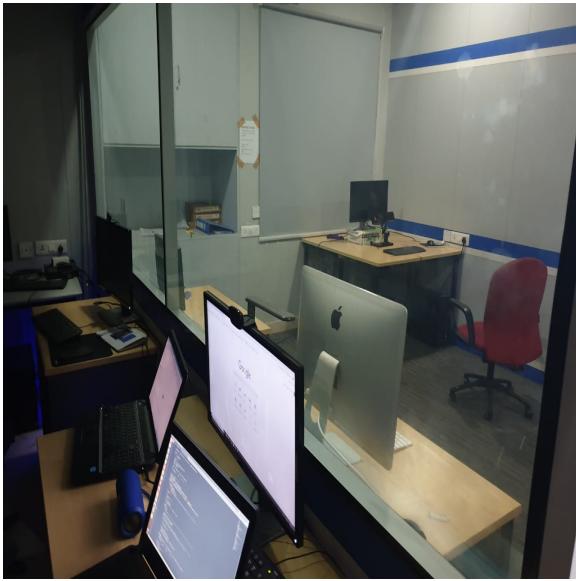
iOS 4

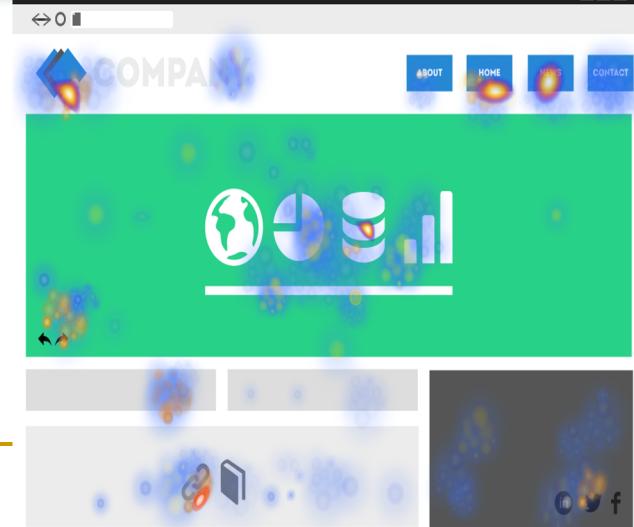
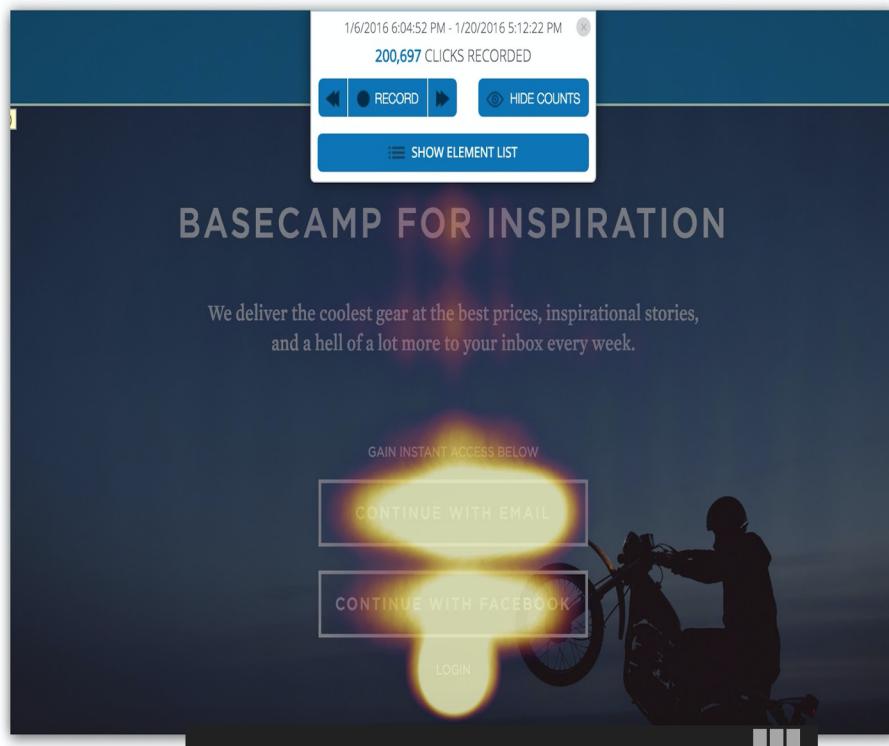
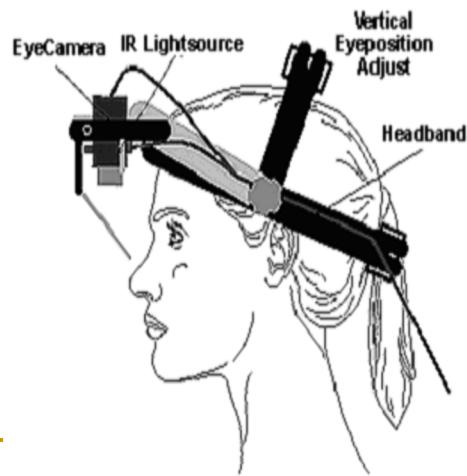
Usability Labs





Virtual Human Interaction Lab (@ SERC, IIITH)





Usability Guidelines

Usability Guidelines are statements by which to determine an action of usefulness. A Usability guideline aims to streamline particular processes according to a set routine or sound practice. By definition, a guideline is never mandatory. Guidelines are not binding and are not enforced.



NN/g



Usability guideline examples

- In a form, telephone number entry is restricted to a dial-pad (i.e. numbers only)
- User can select to reveal or hide password as they type, during signup or sign-in (e.g. by toggling a ‘reveal’ or ‘hide’ control)
- If app requests sign-up, the user can choose to continue as a guest
- First-time user has multiple sign-up options (e.g.,username/password and social sign-up facebook.com, google, twitter)



Factory pattern

Acknowledgement: Head-first design patterns

461 Pizza Store

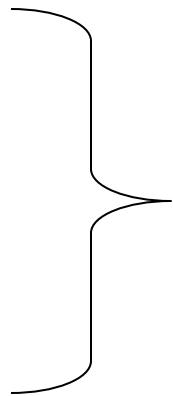
```
public class PizzaStore{  
    Pizza orderPizza() {  
  
        Pizza pizza = new Pizza();  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

If you need more than one type...

```
Pizza orderPizza(String type) {  
    Pizza pizza ;  
  
    if (type.equals("cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals("greek")) {  
        pizza = new GreekPizza();  
    } else if (type.equals("pepperoni")) {  
        pizza = new PepperoniPizza();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

still more...

```
Pizza orderPizza(String type) {  
    Pizza pizza;  
    if (type.equals("cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals("greek")) {  
        pizza = new GreekPizza();  
    } else if (type.equals("pepperoni")) {  
        pizza = new PepperoniPizza();  
    } else if (type.equals("chicken")) {  
        pizza = new ChickenPizza();  
    } else if (type.equals("veggie")) {  
        pizza = new VeggiePizza();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```



1. Dealing with which concrete class is being instantiated
2. Changes are preventing `orderPizza()` from being closed for modification

Building a simple Pizza factory

```
Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

```
public class SimplePizzaFactory{  
    public Pizza createPizza (String type) {  
        Pizza pizza = null;  
  
        if (type.equals("cheese")) {  
            pizza = new CheesePizza();  
        }  
        else if (type.equals("pepperoni")) {  
            pizza = new PepperoniPizza();  
        }  
        else if (type.equals("chicken")) {  
            pizza = new ChickenPizza();  
        }  
        else if (type.equals("veggie")) {  
            pizza = new VeggiePizza();  
        }  
    }  
}
```

Dumb Question...is it really?

Hmmm...

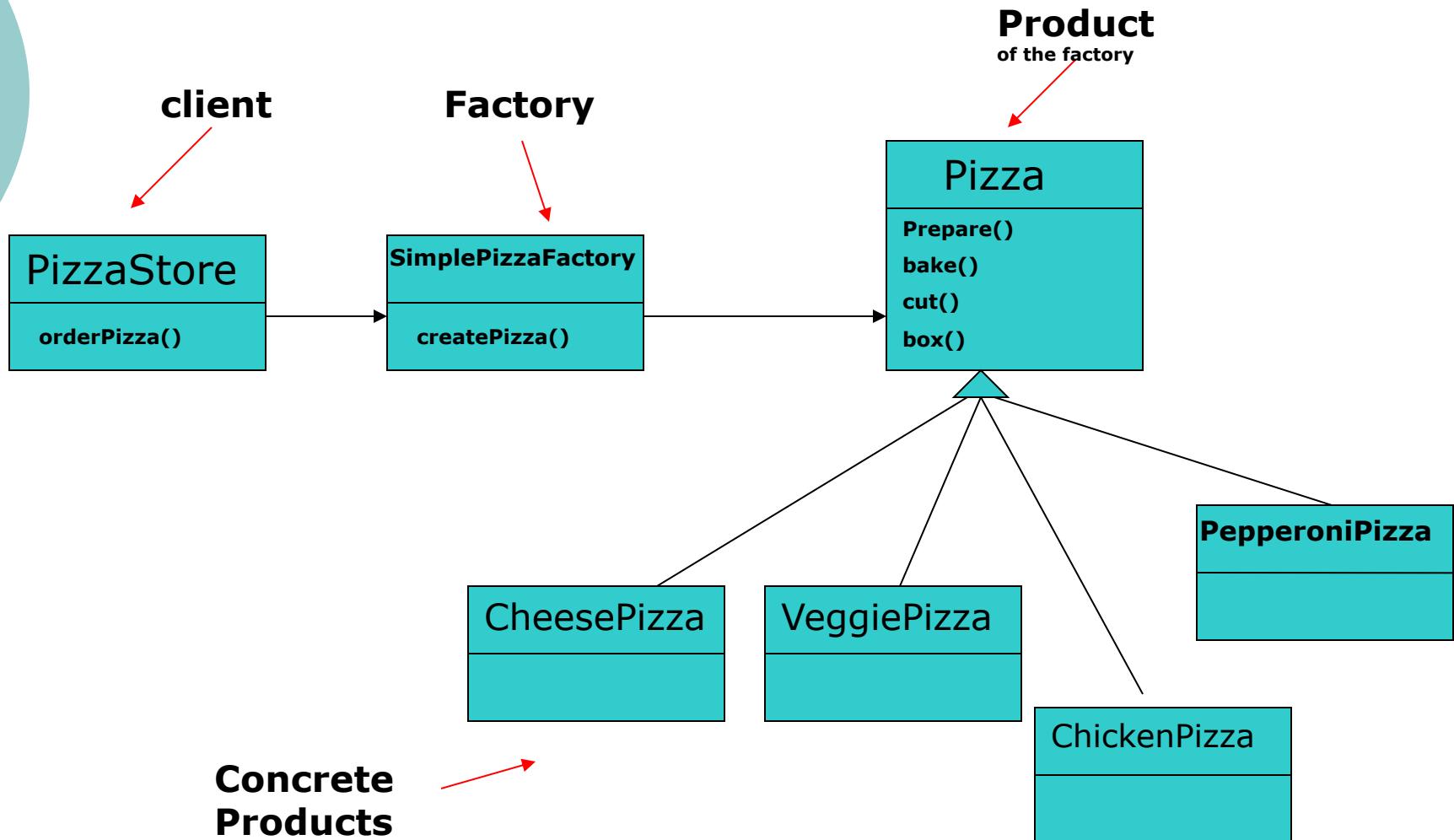
What's the advantage of this?

1. Simple pizza factory may have many clients that use the creation in different ways
2. Easy to remove concrete instantiations from the client code

Changing the client class

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
  
    public PizzaStore (SimplePizzaFactory factory) {  
        this.factory = factory;  
    }  
  
    public Pizza orderPizza (String type) {  
        Pizza pizza;  
  
        pizza = factory.createPizza(type);  
  
        pizza.prepare()  
        pizza.bake()  
        pizza.cut()  
        pizza.box()  
        return pizza;  
    }  
}
```

Simple Factory



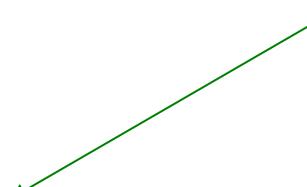
Revisiting - 461 Pizza Store

```
public class PizzaStore{  
    Pizza orderPizza() {  
  
        Pizza pizza = new Pizza();  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

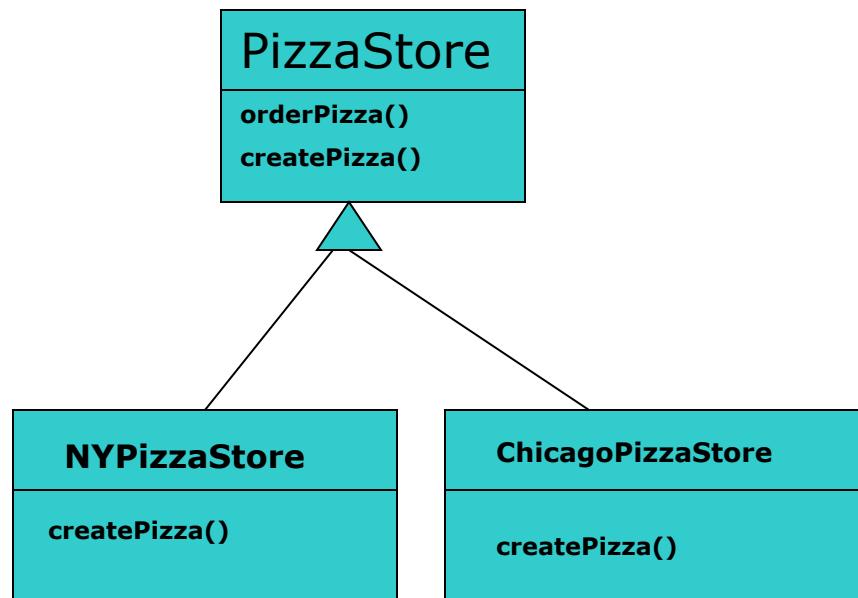
A framework for the pizza store

```
public abstract class PizzaStore {  
  
    public Pizza orderPizza (String type) {  
        Pizza pizza;  
  
        pizza = createPizza(type);  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
  
    abstract Pizza createPizza (String type);  
}
```

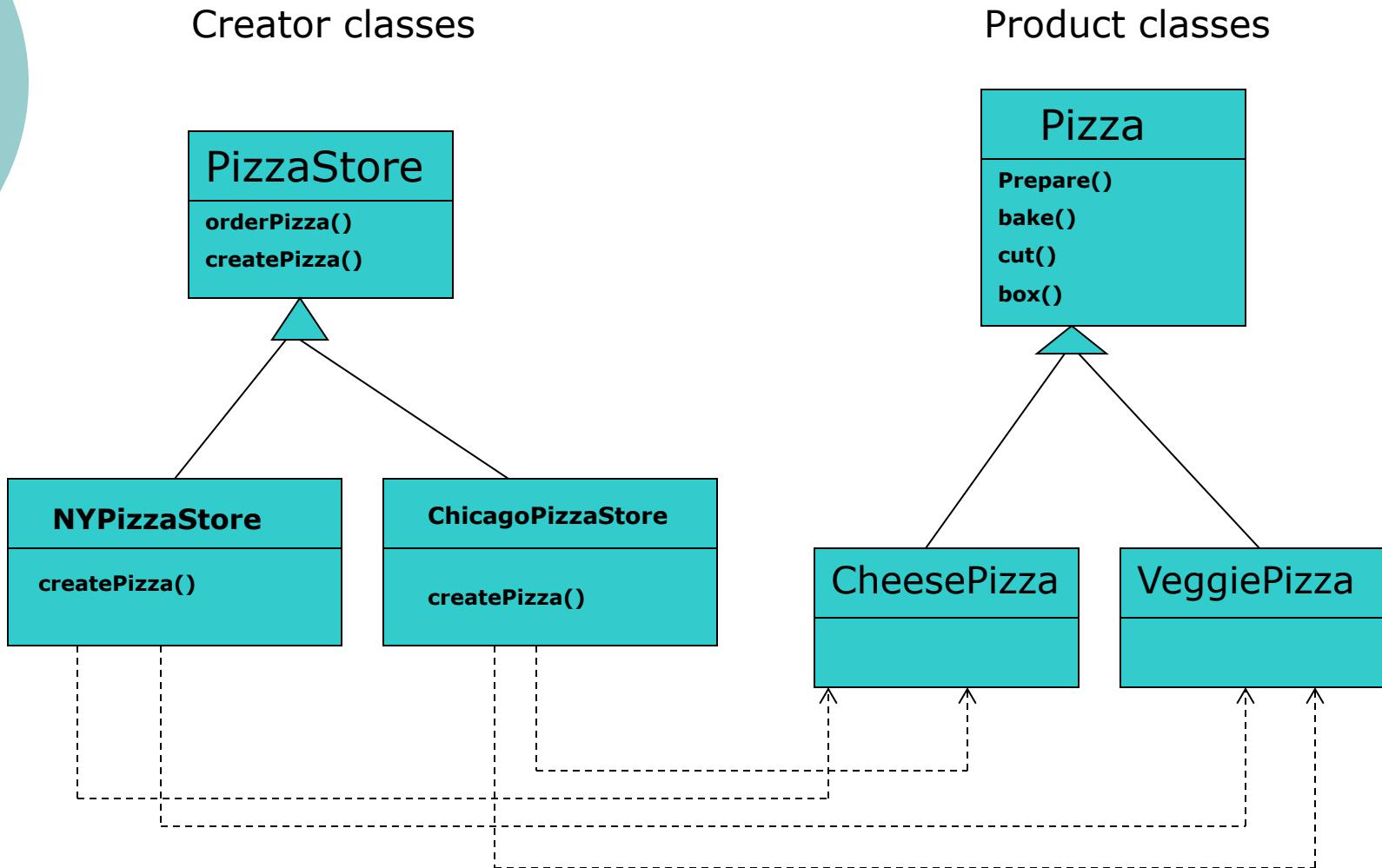
Factory
method is
now
abstract



Allowing the subclasses to decide (creator classes)

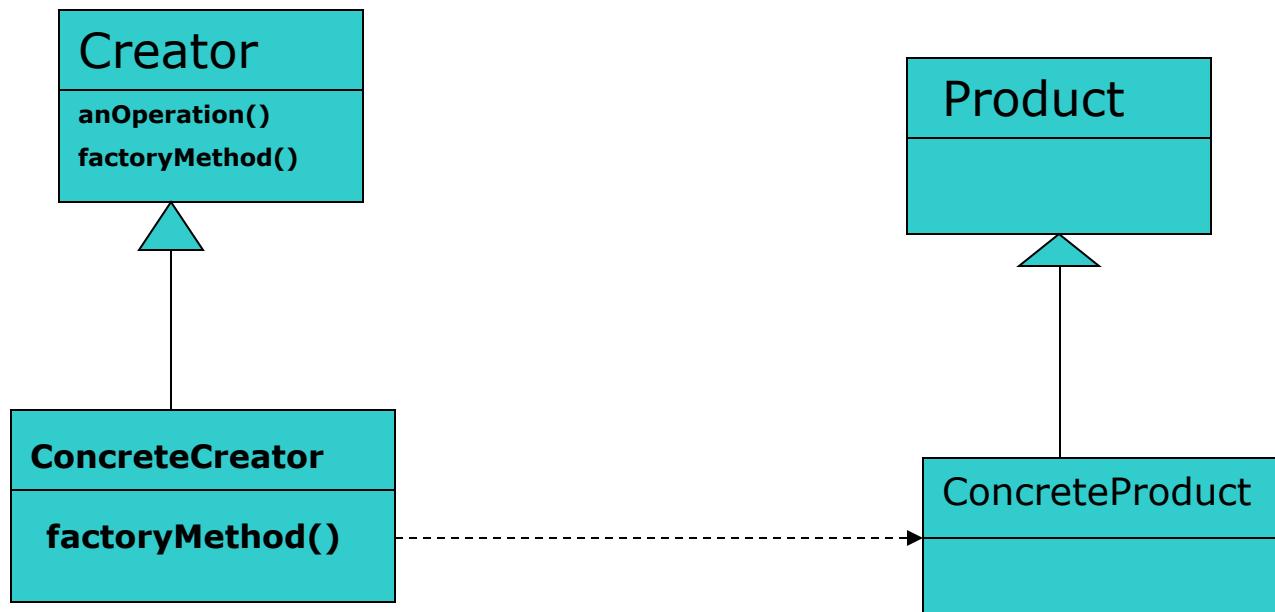


Factory method



Factory method defined

- Factory method pattern defines an interface for creating an object, but lets the subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses





Flyweight pattern

Acknowledgement: Head-first design patterns

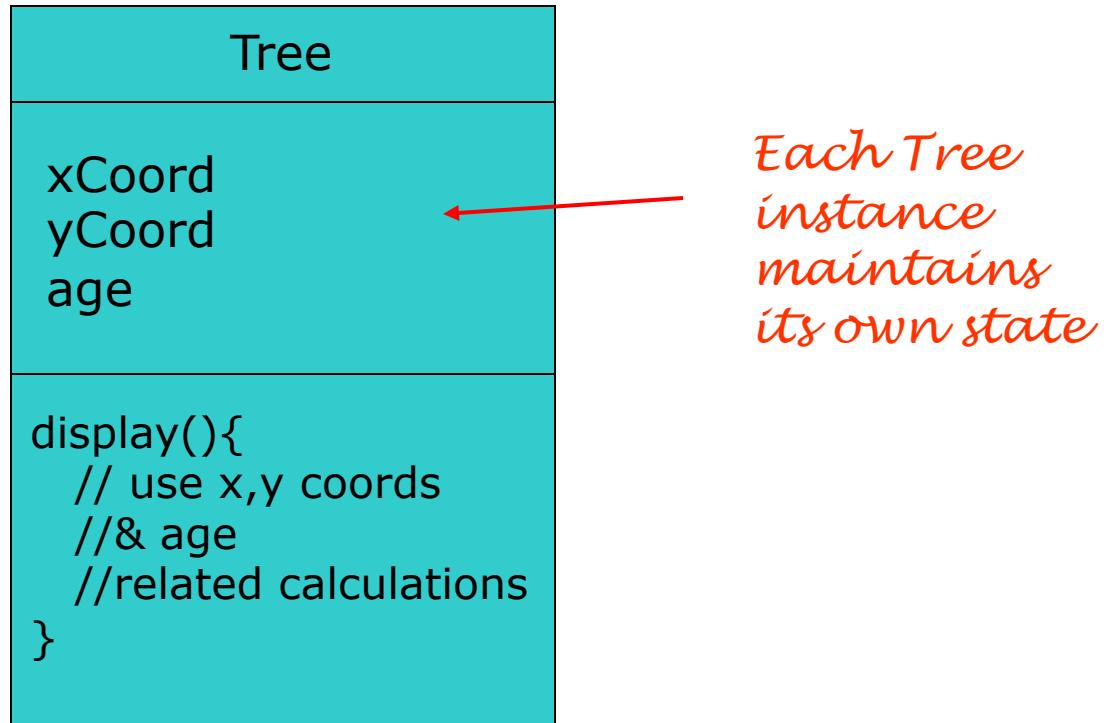
When do we use it?

- When one instance of a class can be used to provide many “virtual instances”

Example scenario

- Wish to add trees as objects in a landscape design
- They just contain x,y location and draw themselves dynamically depending on the age of the tree
- User may wish have lots of trees in a particular landscape design

Tree class



What happens if a 100000 tree objects are created?

Flyweight pattern

- If there is only one instance of Tree and client object maintains the state of ALL the Trees, then it's a **flyweight**

TreeManager
treeArray
<pre>displayTrees(){ // for all Trees { // get array row // display(x,y,age); } }</pre>

Tree
<pre>display(x,y,age){ // use x, y coords //& age //related calculations }</pre>

Benefits & Drawbacks

Benefits:

- Reduces the number of object instances at runtime, saving memory
- Centralizes state for many “virtual” objects into a single location

Drawbacks:

- Once a flyweight pattern is implemented, single logical instances of the class will not be able to behave independently from other instance.

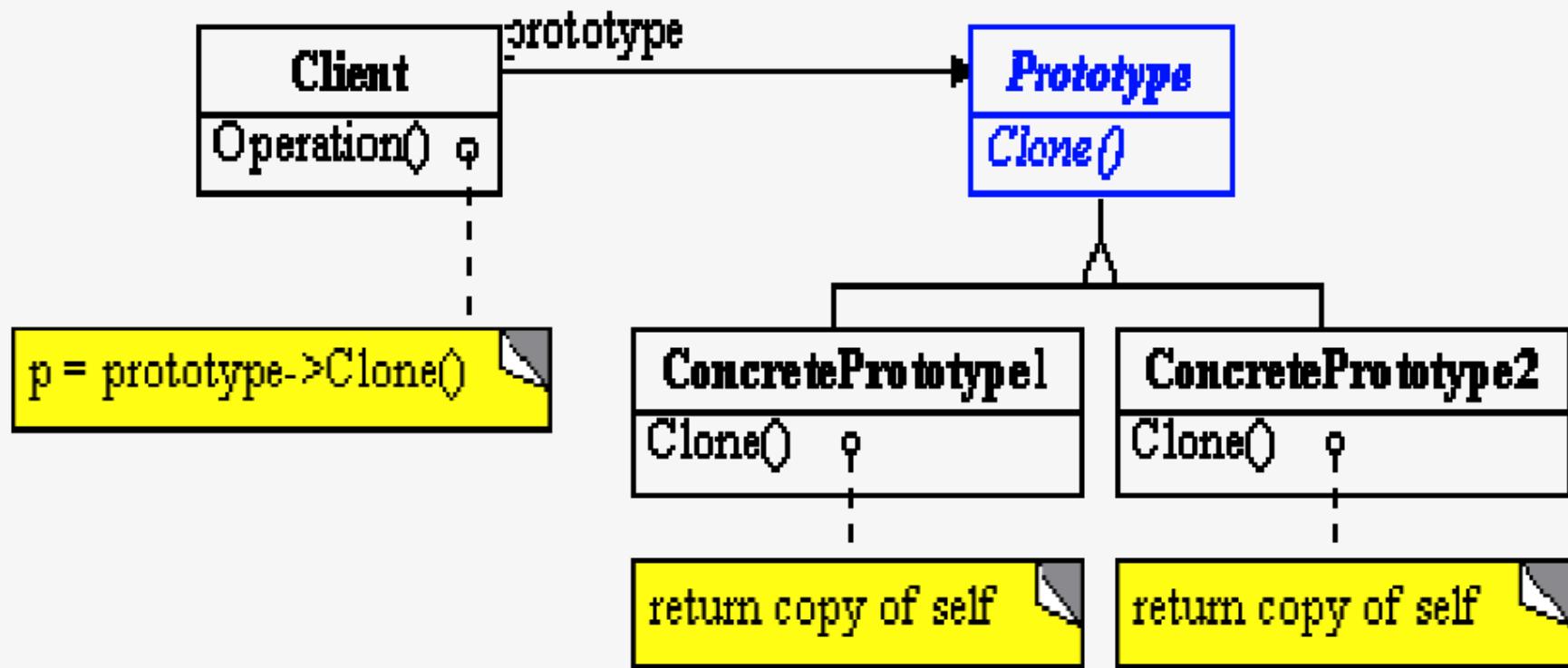


Prototype pattern

Prototype

- Allows for creation of new instances by copying existing instances
(In java, it's done by `clone()` method – shallow copy. If deep copies are needed, they should be handled as `Serializable` objects)
- Client can make new instances without knowing which specific class is being instantiated
- Provide an object like the one it should create
- This template object is a *Prototype* of the ones we want to create
- When we need new object, ask prototype to copy or clone itself

Abstract Structure



Participants

Client

- *Creates a new object by asking a prototype to clone itself*

Prototype

- *Declares an interface for cloning itself*

Concrete Prototype

- *Implements an operation for cloning itself*