ECE4574 – Large-Scale SW Development for Engineering Systems
Lecture 13 – A Large System Design

Creed Jones, PhD

Course Updates

- Sprint 2 is underway
  - Adapt any process changes that identified in Sprint 1
- Homework 2 is due this Friday, October 13
- Quiz 5 is next Monday, October 15
  - covers lectures 12-14
  - open 7 PM to 1 AM

Today's Objectives

Some thoughts on large software systems design

- decomposition
- WBS
- middleware

An example of an online photo retouching system

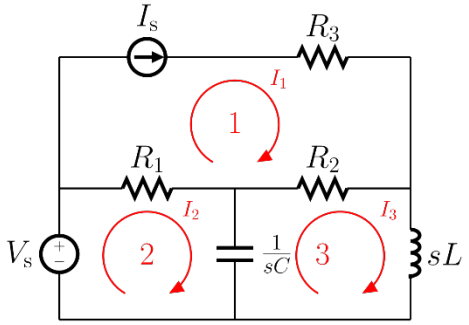
Architecture of a Component

- Component-based architecture
- Defining a component
- Designing a component

SOME THOUGHTS ON LARGE SYSTEMS DESIGN

I claim that the fundamental skill of an engineer is breaking up a large problem into smaller, more easily solvable problems - **decomposition**

- Math
  - Integration by parts: $\int u\,dv = uv - \int v\,du$
  - Taylor series expansion: $f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots$
- Circuits
  - Mesh analysis:



- Mechanical / Product Design
  - Partitioning a product into subsystems

This is especially true in software systems – most tractable solutions come from solving pieces of the overall problem in interconnectable ways

- The overall list of requirements is broken into groups that are highly cohesive and loosely coupled
- Each of those requirement groups will be met by a subsystem
  - Then, subsystems are broken down into components
  - Again, striving for high cohesion and low coupling
    - Components are made up of objects
    - Classes are defined to produce the required objects

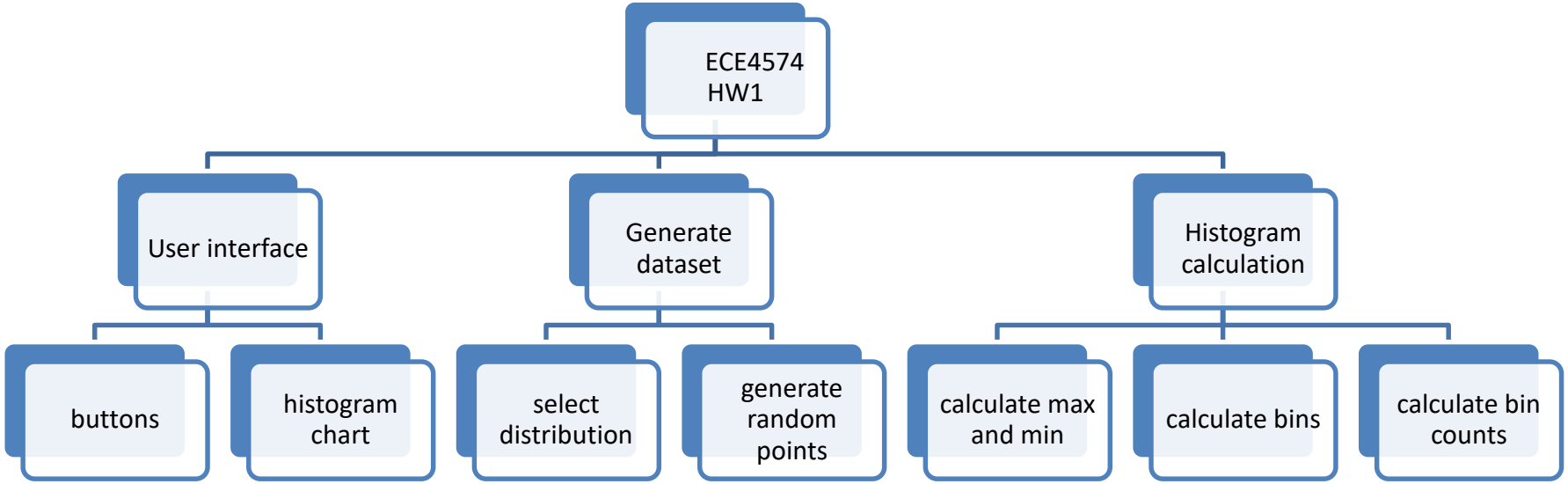
- Recall:
  - Cohesion is the degree to which all pieces support a common purpose
  - Coupling is the number of connections (through proper interfaces)

Generally the architecture of a large software system will look something like the *work breakdown structure* of the project

- A WBS is a hierarchical arrangement of the tasks required to complete some project
  - The top level is the entire project
  - We then split it into subtasks and sub-subtasks until we reach the desired level of detail
  - At the lowest level, try to achieve similar levels of effort
- Since a task at the lowest level will be accomplished by an individual or small team, a logical partition into components and subsystems will follow the work breakdown structure

Here is a possible work breakdown structure for a program that generates and graphs some random data

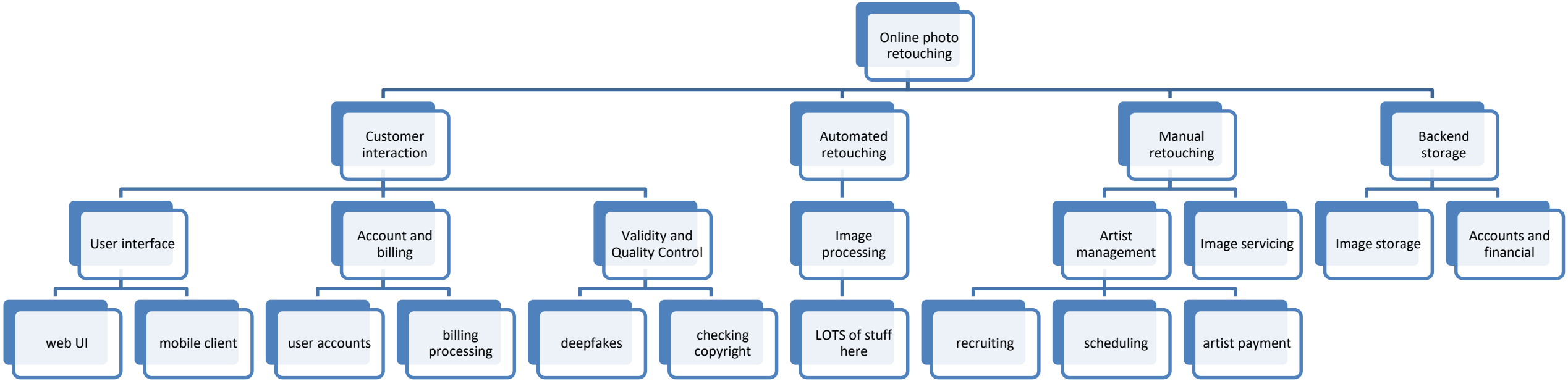
- Upon startup, create a sample of 10,000 data points from a normal (Gaussian) distribution
- Automatically calculate the min and max values of the data
- Divide the range of the data into 20 equally spaced bins
- Plot the frequency distribution (the histogram) in these bins as a bar chart
- Provide three radio buttons to select other distributions
- When the radio button is pressed, repeat the histogram process with the newly selected probability distribution



Let's imagine a software system to allow customers to submit their own photos for professionals to retouch and enhance

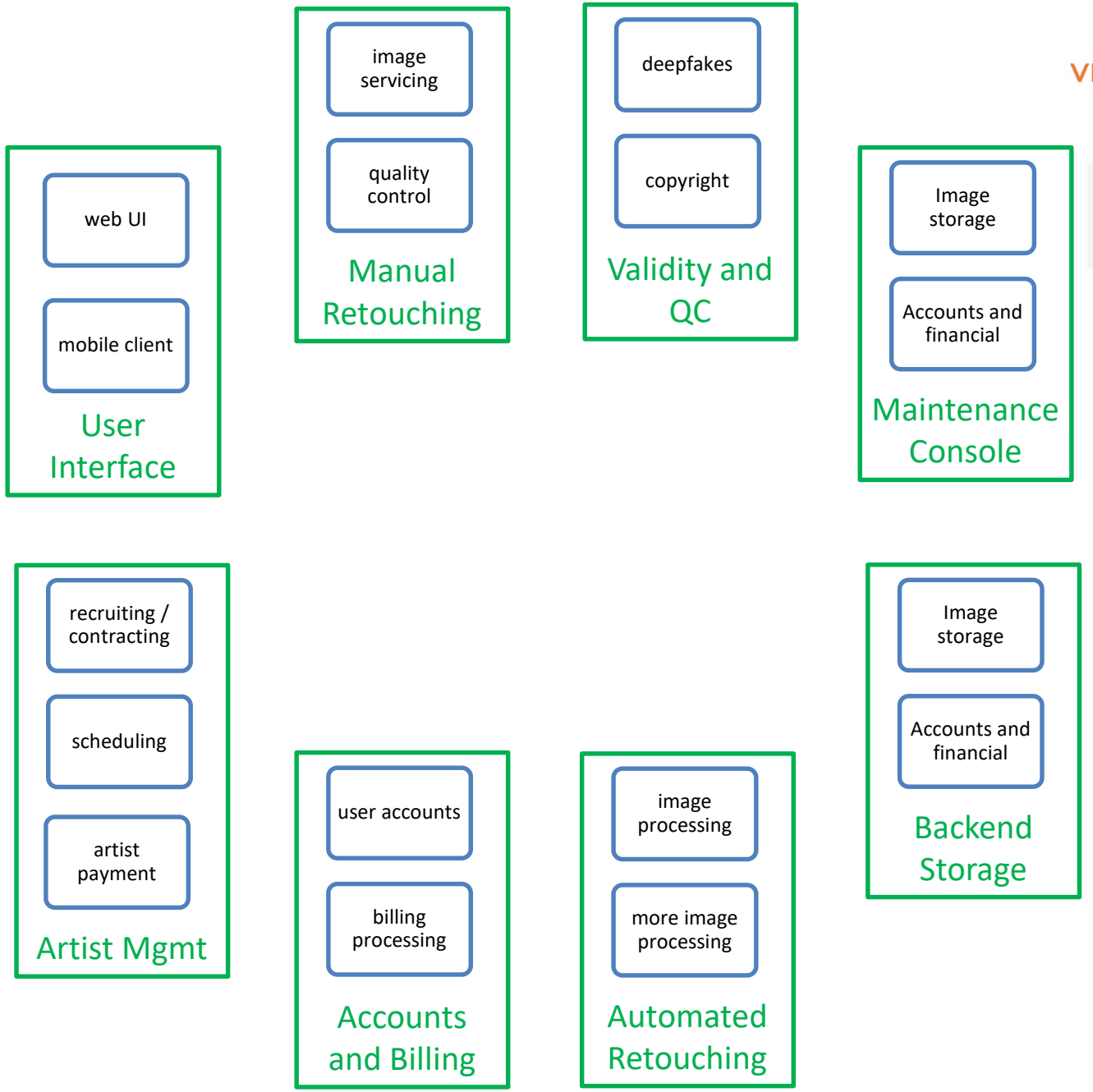
- Customers can either set up an account, or shop as guest and pay with card or PayPal
- They can upload images in one of several graphical formats
- Customers can either:
  - choose from a set of usual retouchings ("remove red-eye", "sharpen", "blur the background")
  - or specify their own custom retouchings ("remove uncle Bob", "hide the wrinkles under my eyes"
- Certain illegal or unethical modifications must be checked for
  - no deepfaking
  - no editing of copyrighted images, unless authorized by copyright holder
- Some retouchings are done automatically by an algorithm; others are done manually
- If the customer is not happy, they get one round of further revisions free
- Image is returned by email, or to public cloud drive
  - also offer cloud storage of photos
- Customers get a discount on photo printing at their local retail outlet

- Customers can either set up an account, or shop as guest and pay with card or PayPal
- They can upload images in one of several graphical formats
- Customers can either:
  - choose from a set of usual retouchings ("remove red-eye", "sharpen", "blur the background")
  - or specify their own custom retouchings ("remove uncle Bob", "hide the wrinkles under my eyes"
- Certain illegal or unethical modifications must be checked for
  - no deepfaking
  - no editing of copyrighted images, unless authorized by copyright holder
- Some retouchings are done automatically by an algorithm; others are done manually
- If the customer is not happy, they get one round of further revisions free
- Image is returned by email, or to public cloud drive
  - also offer cloud storage of photos
- Customers get a discount on photo printing at their local retail outlet



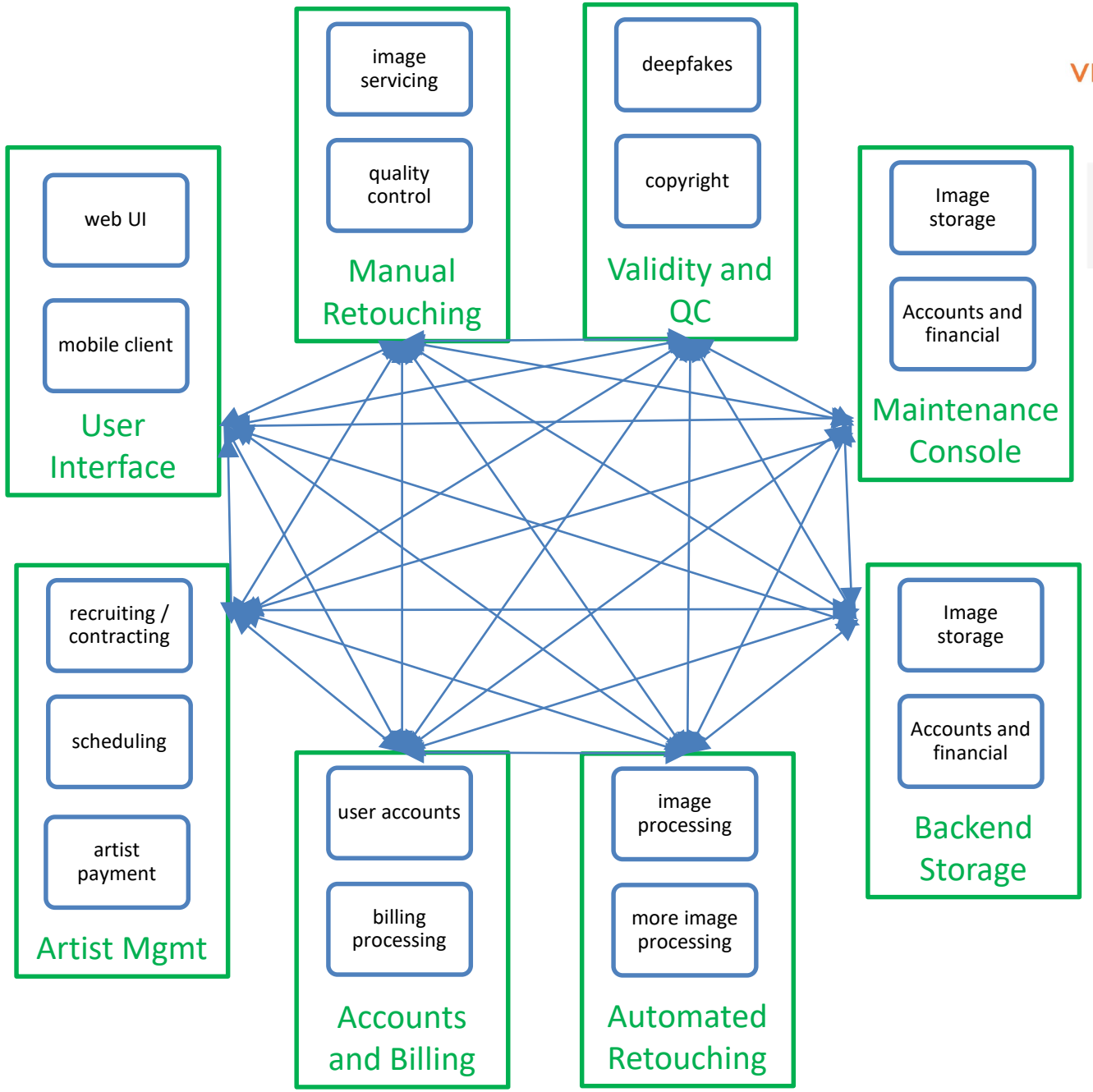
In my first cut of the system design, I define subsystems from the large groups of system requirements
Now the question becomes – how are they connected?

We must choose an overall system architecture



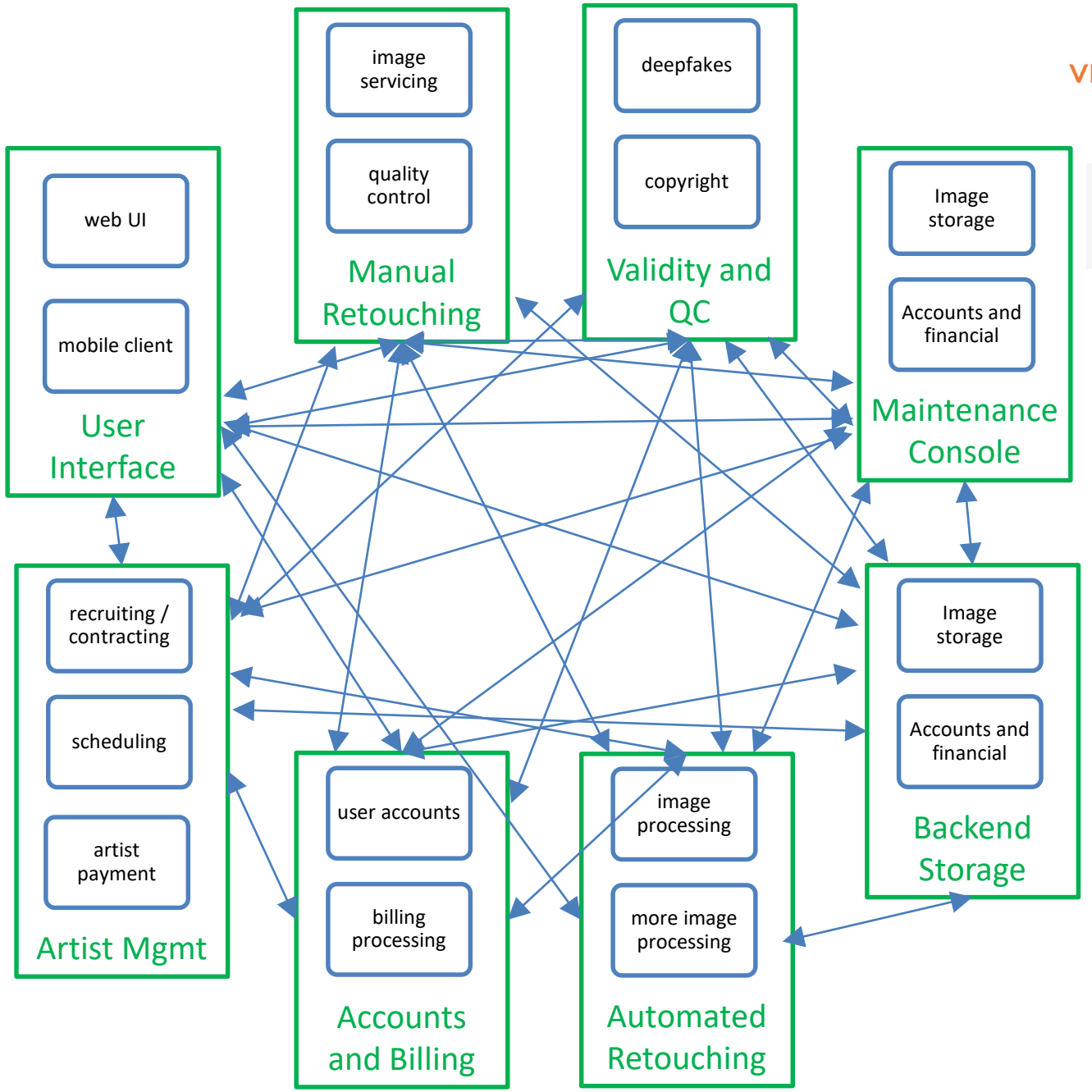
Perhaps each subsystem communicates with every other subsystem directly, through its defined interface?

Lots of connections



Perhaps each subsystem communicates with every other subsystem directly, through different interfaces?

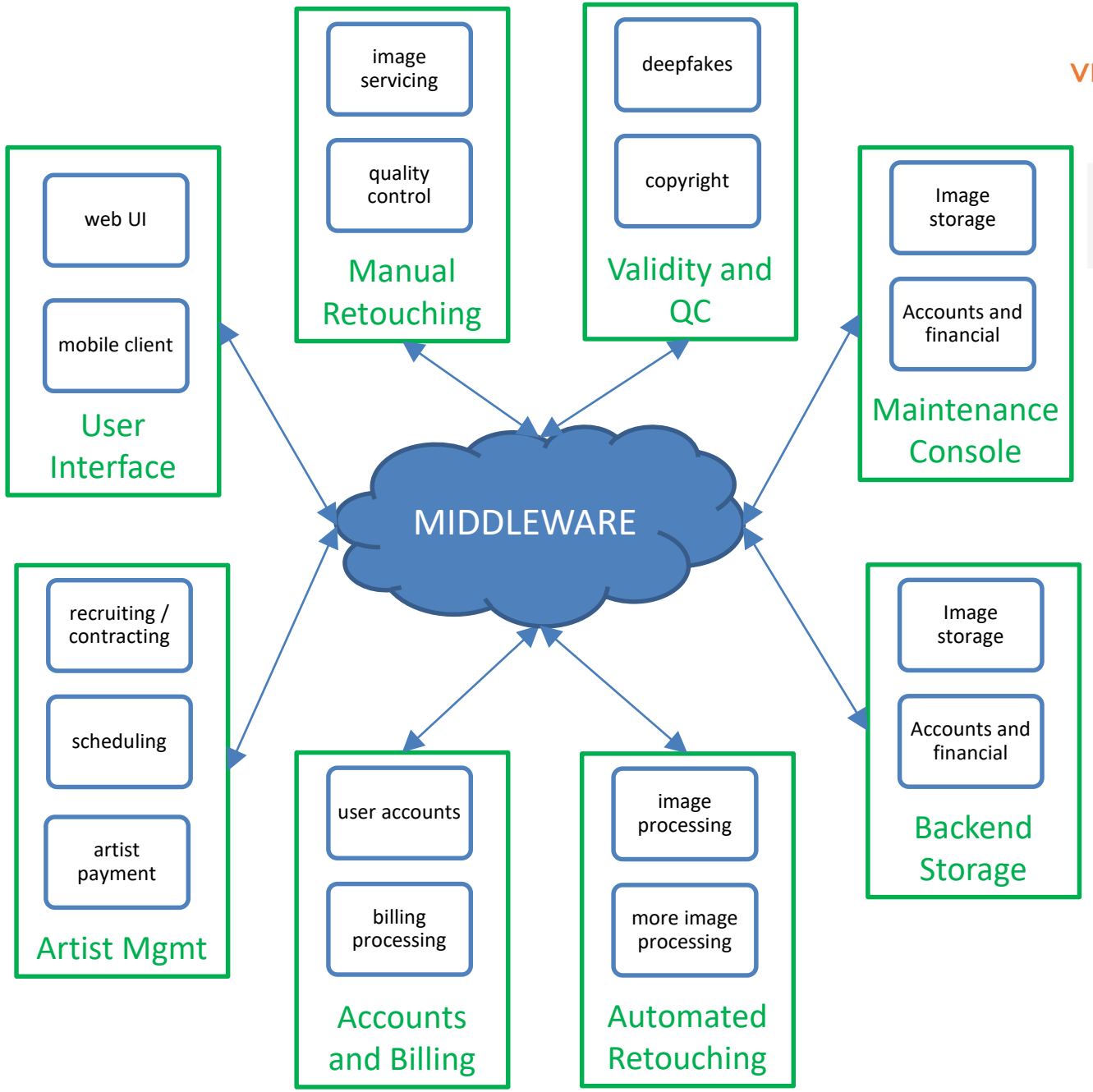
I hope not!



Perhaps there is some software subsystem in the center, to handle intercommunication?

This subsystem could also handle sequencing, logging and other responsibilities

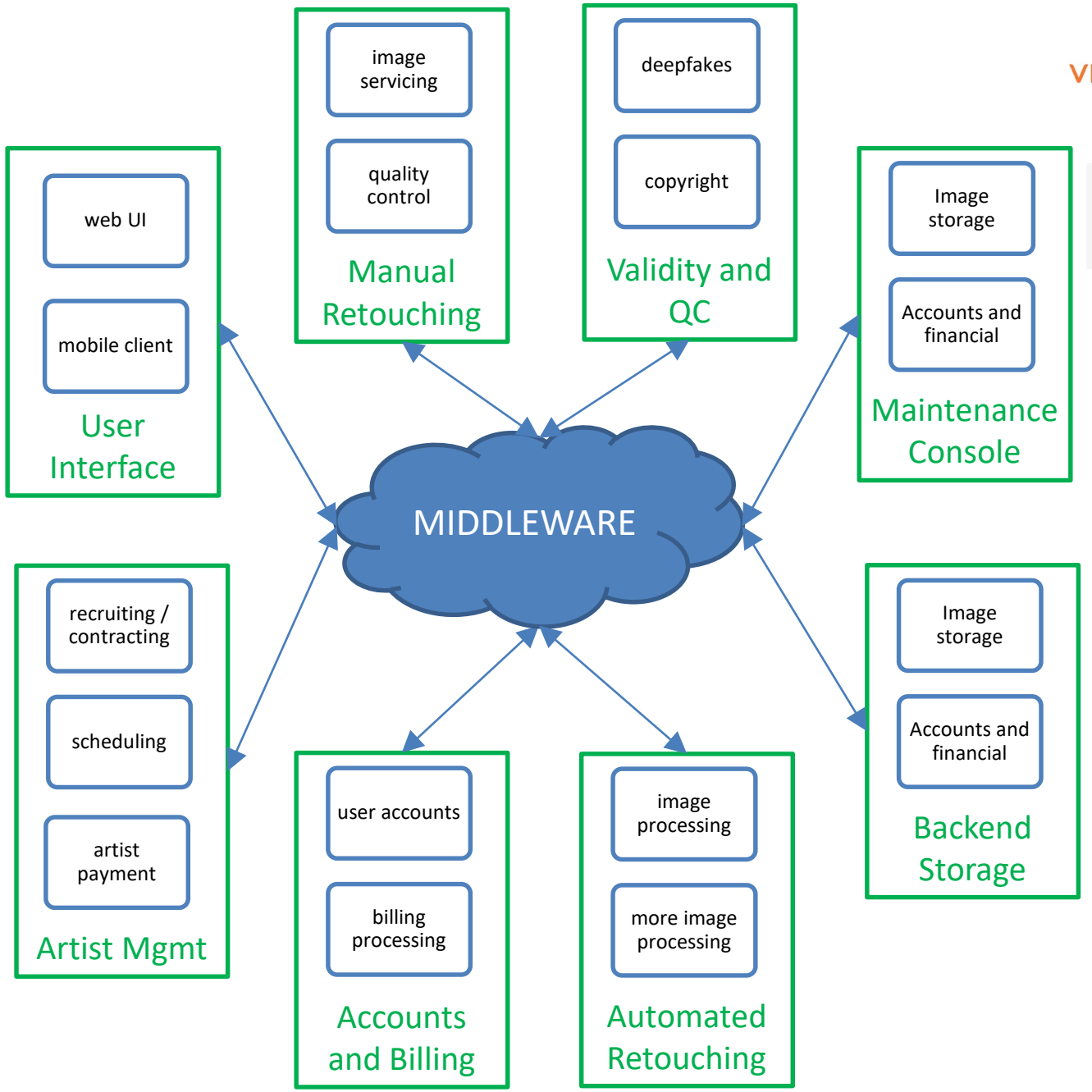
This is what "middleware" does



Let's examine how a Message Broker might serve as the middleware for this application

Requirements

- Customers can either set up an account, or shop as guest and pay with card or PayPal
- They can upload images in one of several graphical formats
- Customers can either:
  - choose from a set of usual retouchings ("remove red-eye", "sharpen", "blur the background")
  - or specify their own custom retouchings ("remove uncle Bob", "hide the wrinkles under my eyes"
- Certain illegal or unethical modifications must be checked for
  - no deepfaking
  - no editing of copyrighted images, unless authorized by copyright holder
- Some retouchings are done automatically by an algorithm; others are done manually
- If the customer is not happy, they get one round of further revisions free
- Image is returned by email, or to public cloud drive
  - also offer cloud storage of photos
- Customers get a discount on photo printing at their local retail outlet



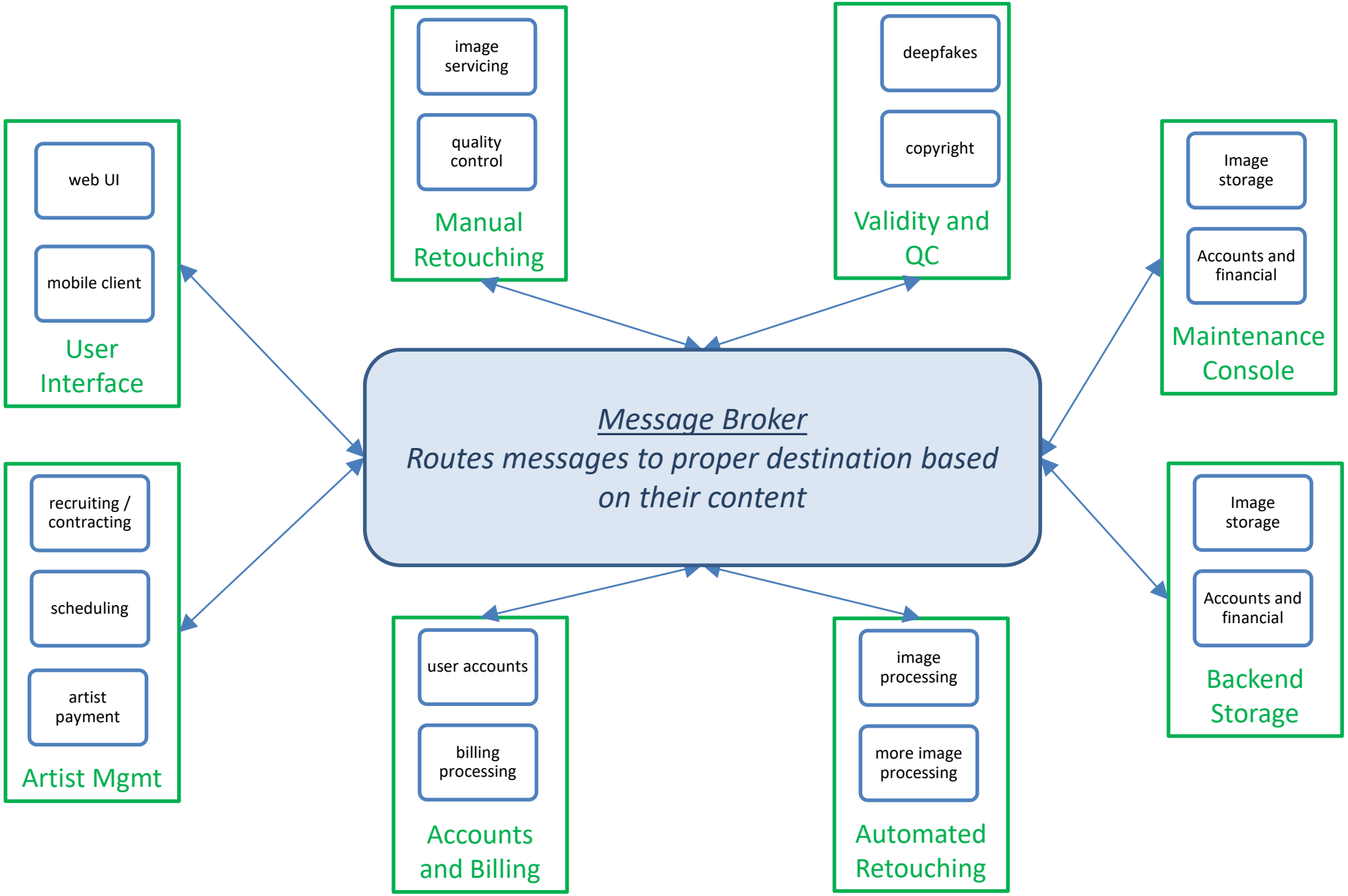
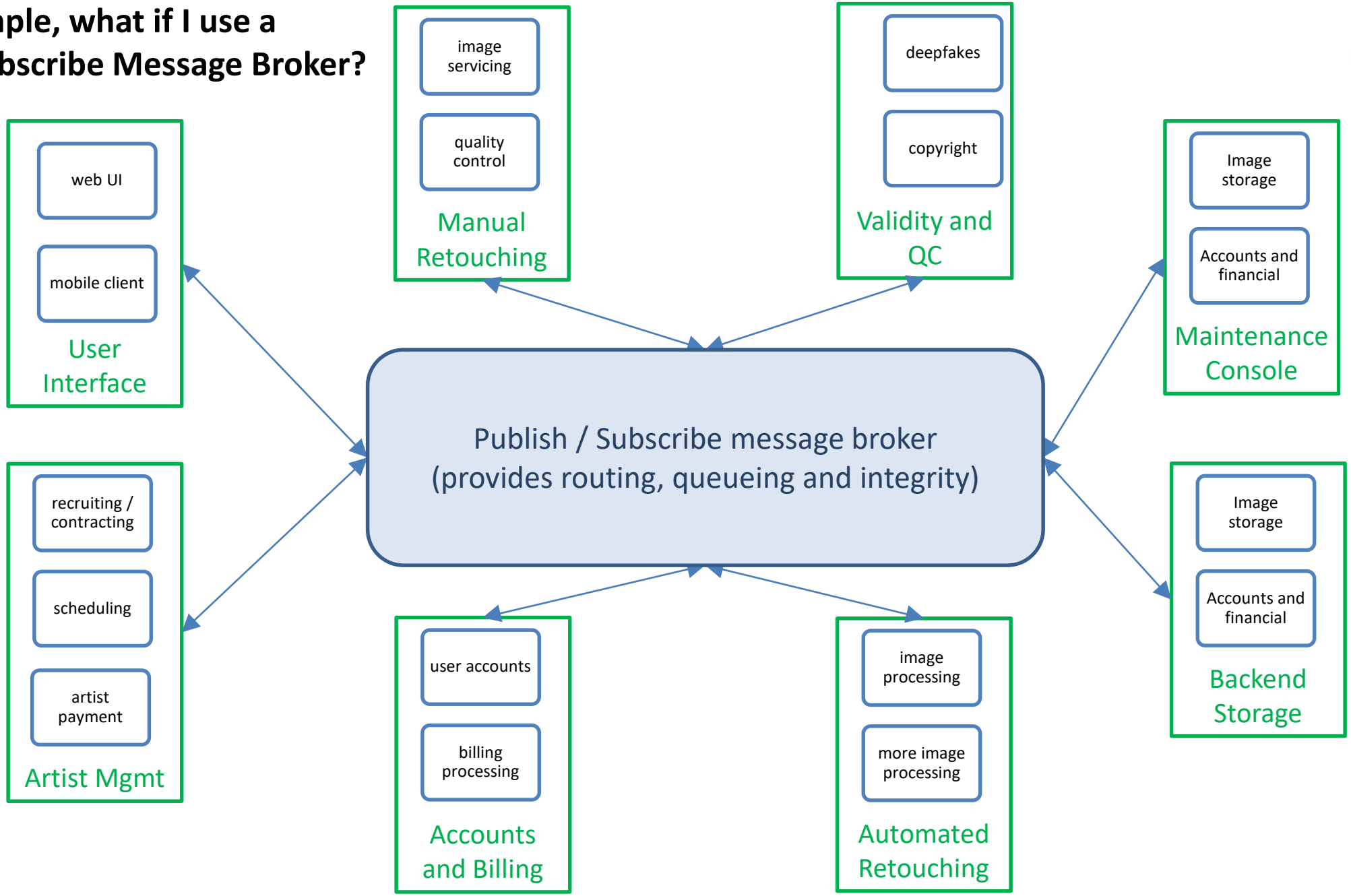


image servicing
quality control
Manual Retouching
deepfakes
copyright
Validity and QC
web UI
mobile client
User Interface
Image storage
Accounts and financial
Maintenance Console
Message Broker
Routes messages to proper destination based on their content
recruiting / contracting
scheduling
artist payment
Artist Mgmt
Image storage
Accounts and financial
Backend Storage
user accounts
billing processing
Accounts and Billing
image processing
more image processing
Automated Retouching

As an example, what if I use a Publish/Subscribe Message Broker?



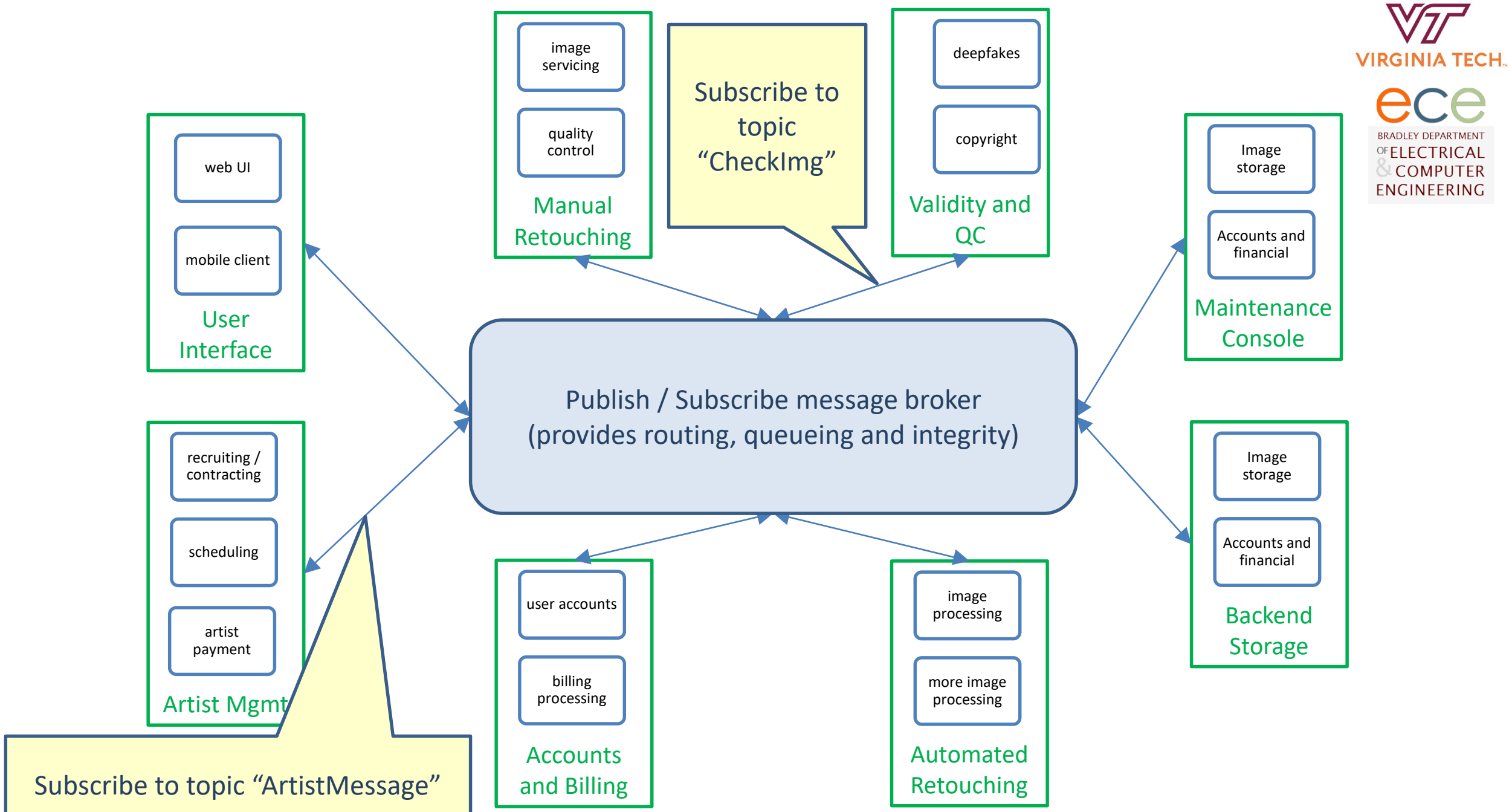


image servicing
quality control
Manual Retouching
Subscribe to topic "CheckImg"
deepfakes
copyright
Validity and QC
web UI
mobile client
User Interface
Image storage
Accounts and financial
Maintenance Console
Publish / Subscribe message broker
(provides routing, queueing and integrity)
recruiting / contracting
scheduling
artist payment
Artist Mgmt
Image storage
Accounts and financial
Backend Storage
user accounts
billing processing
Accounts and Billing
image processing
more image processing
Automated Retouching
Subscribe to topic "ArtistMessage"

We can start to define topics that each component will need to subscribe to; note that the workflow is determined by two things: who subscribes to each topic and what they publish in response

Topic	Publisher(s)	Subscriber(s)	Notes
ArtistMessage	Maintenance, Validity and QC, *Scheduling**	Artist Mgmt, Backend	Need a scheduling component
CheckImg	*Scheduling*, Artist Mgmt	Validity and QC, Backend	Images are checked then stored
UserAccountMsg	UI, Maintenance	Accounts and Billing, Backend	
ArtistWork	*Scheduling**, Backend, Validity and QC	Manual Retouching, Backend	Work components get images from the backend
ImgProcWork	*Scheduling**, Backend, Validity and QC	Automated Retouching, Backend	
<etc…>	<etc…>	<etc…>	

Note how this supports OOD – each component can be designed and implemented to fulfil its charter

Topic	Publisher(s)	Subscriber(s)	Notes
CheckImg	*Scheduling*, Artist Mgmt	Validity and QC, Backend	Images are checked then stored

- The Validity and QC component will receive messages from the Scheduling component (asking for an image to be checked) and possibly from Artist Mgmt (for verifying or rechecking)
- It will publish to the topics:
  - ImageToStore, which the Backend Storage component subscribes to
  - Logging, which keeps a record of all transactions in the Backend Storage
- That might be it!
  - Though, if we add more complexity, it's just a matter of publishing to, or subscribing to, additional topics

COMPONENT ARCHITECTURE

A *component-based architecture* is built from pieces that use well-defined interfaces to communicate

- Reusability helps us achieve the big goals of shorter development time, lower cost and higher quality (assuming that existing components are better proved)
- Shorter development time lowers cost but also (and often more importantly) reduces the *time-to-market*

- We have discussed the decomposition of a system into large pieces that accomplish parts of the overall system backlog
- We call these components; let's put together a more rigorous definition for a component, then discuss how to architect and design one

A software component is a portion of a system that provides a cohesive set of services, through well-defined interfaces

- Reusable – Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task.
- Replaceable – Components may be freely substituted with other similar components.
- Not context specific – Components are designed to operate in different environments and contexts.
- Extensible – A component can be extended from existing components to provide new behavior.
- Encapsulated – A A component depicts the interfaces, which allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state.
- Independent – Components are designed to have minimal dependencies on other components.

Here is a common, detailed approach to decomposing a large system into components

elaborate in this context means to expand and provide more detail

1. Recognize all design classes that correspond to the infrastructure domain.
2. Describe all design classes that are not acquired as reusable components, and specify message details.
3. Identify appropriate interfaces for each component and elaborates attributes and define data types and data structures required to implement them.
4. Describe processing flow within each operation in detail using pseudo code or UML activity diagrams.
5. Describe persistent data sources (databases, files and cloud storage) and identify the classes required to manage them.
6. Develop and *elaborate* behavioral representations for a class or component. This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class.
7. Elaborate deployment diagrams to provide additional implementation detail.
8. Demonstrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and OS environment.
9. Final architectural decisions can be made by using established design principles and guidelines. Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model.

https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm

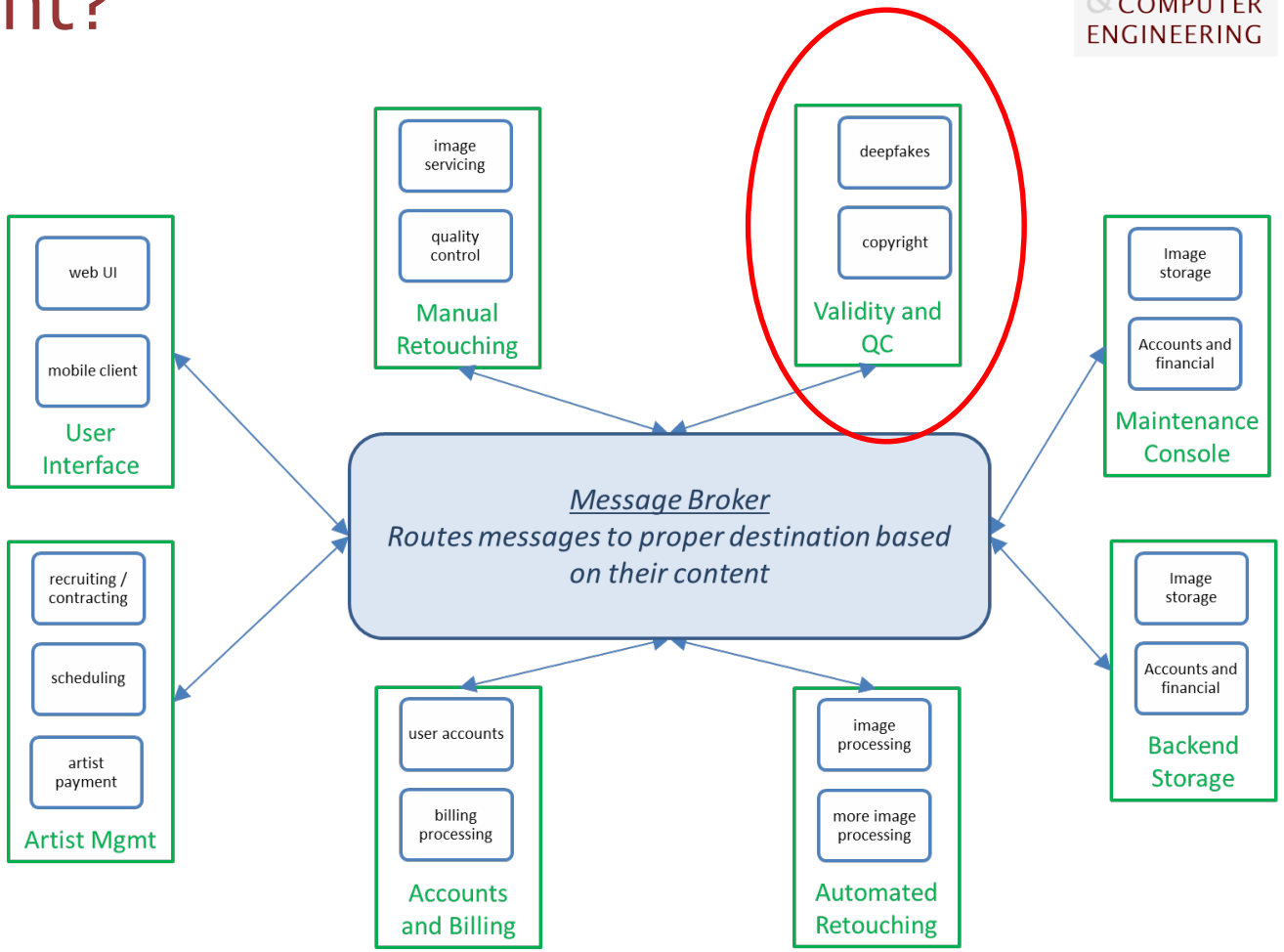
Some of these steps concern the internal architecture of the components themselves

1. Recognize all design classes that correspond to the infrastructure domain.
2. Describe all design classes that are not acquired as reusable components, and specify message details.
3. Identify appropriate interfaces for each component and elaborates attributes and define data types and data structures required to implement them.
4. Describe processing flow within each operation in detail using pseudo code or UML activity diagrams.
5. Describe persistent data sources (databases, files and cloud storage) and identify the classes required to manage them.
6. Develop and *elaborate* behavioral representations for a class or component. This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class.
7. Elaborate deployment diagrams to provide additional implementation detail.
8. Demonstrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and OS environment.
9. Final architectural decisions can be made by using established design principles and guidelines. Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model.

https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm

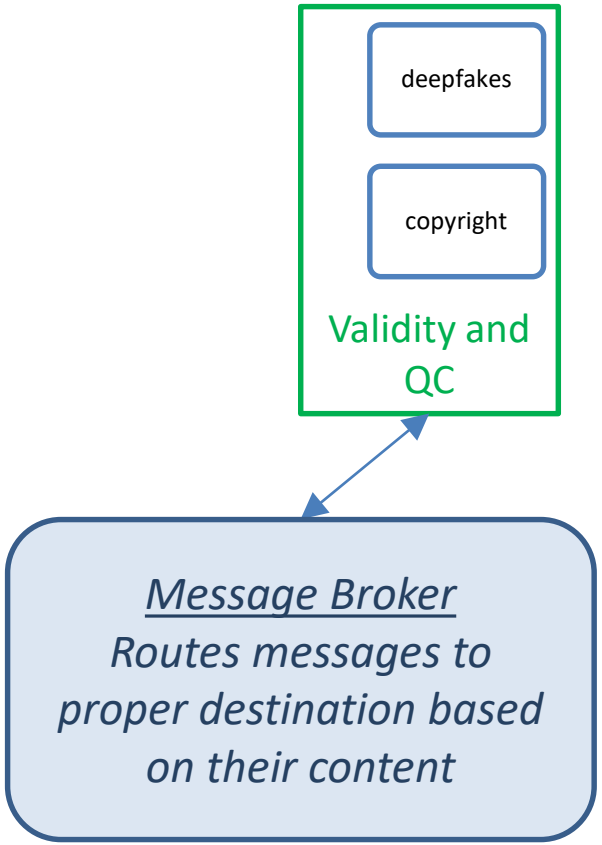
Now that we have defined components, what is the best process to architect, design and implement each component?

- Given:
  - cohesive list of requirements (responsibilities)
  - interface(s) that the component must support
  - non-functional requirements for the system
- We must decide on:
  - the internal architecture
  - specific classes and subcomponents
  - buy vs. build



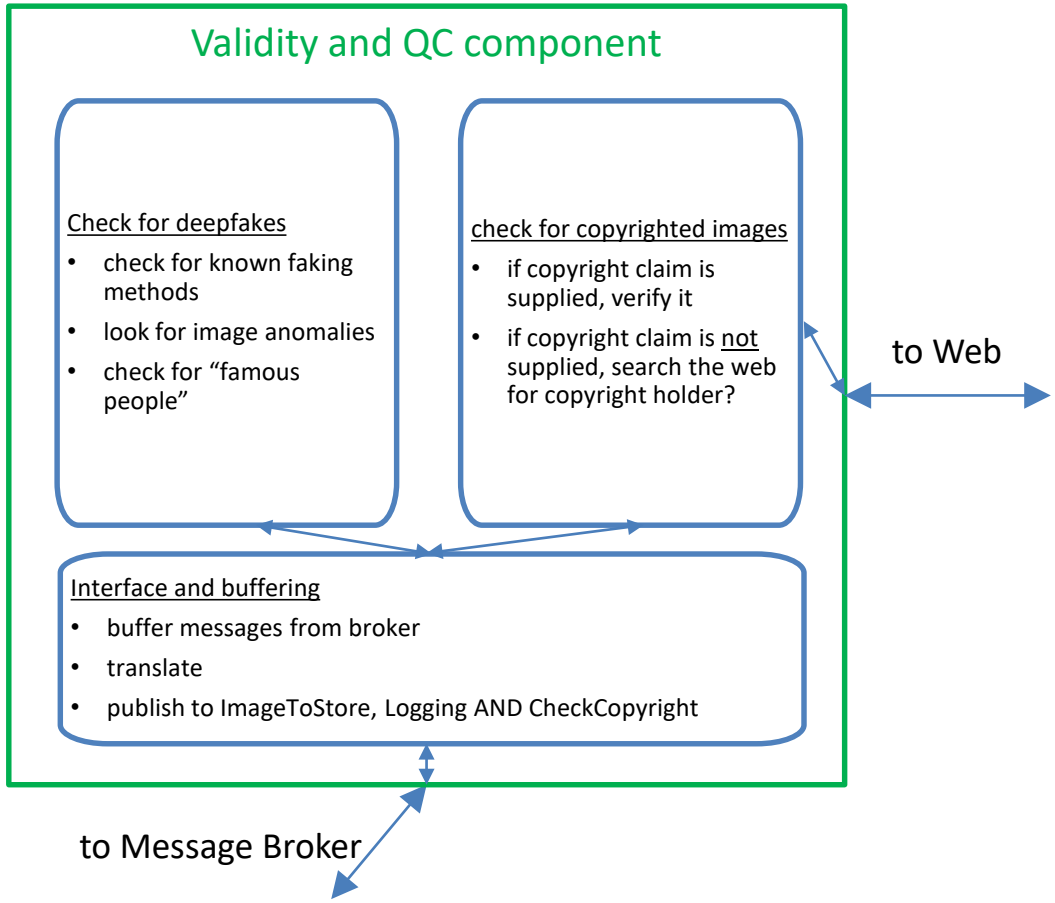
Let's pick up the Validity and QC component from our photo retouching system example and see where the process leads

- The Validity component will receive *CheckImg* messages:
  - from the scheduling component (asking for an image to be checked) and possibly from Artist Mgmt (for verifying or rechecking)
- It will publish to the topics:
  - ImageToStore, which the Backend Storage component subscribes to
  - Logging, which keeps a record of all transactions in the Backend Storage



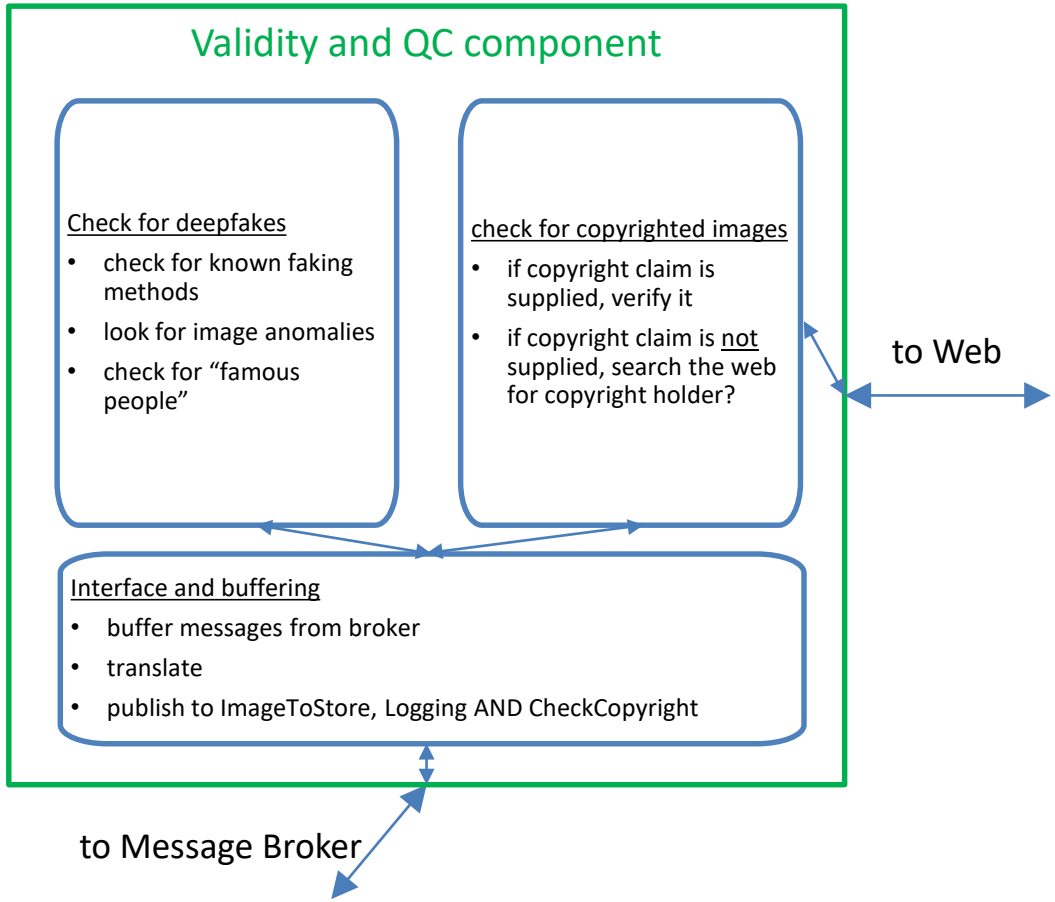
Let's pick up the Validity and QC component from our photo retouching system example and see where the process leads

- The Validity component will receive *CheckImg* messages:
  - from the scheduling component (asking for an image to be checked) and possibly from Artist Mgmt (for verifying or rechecking)
- It will publish to the topics:
  - ImageToStore, which the Backend Storage component subscribes to
  - Logging, which keeps a record of all transactions in the Backend Storage
  - Workflow?
  - CheckLocalImgBuffer, which retains many common deepfakes, famous people and copyrighted images locally

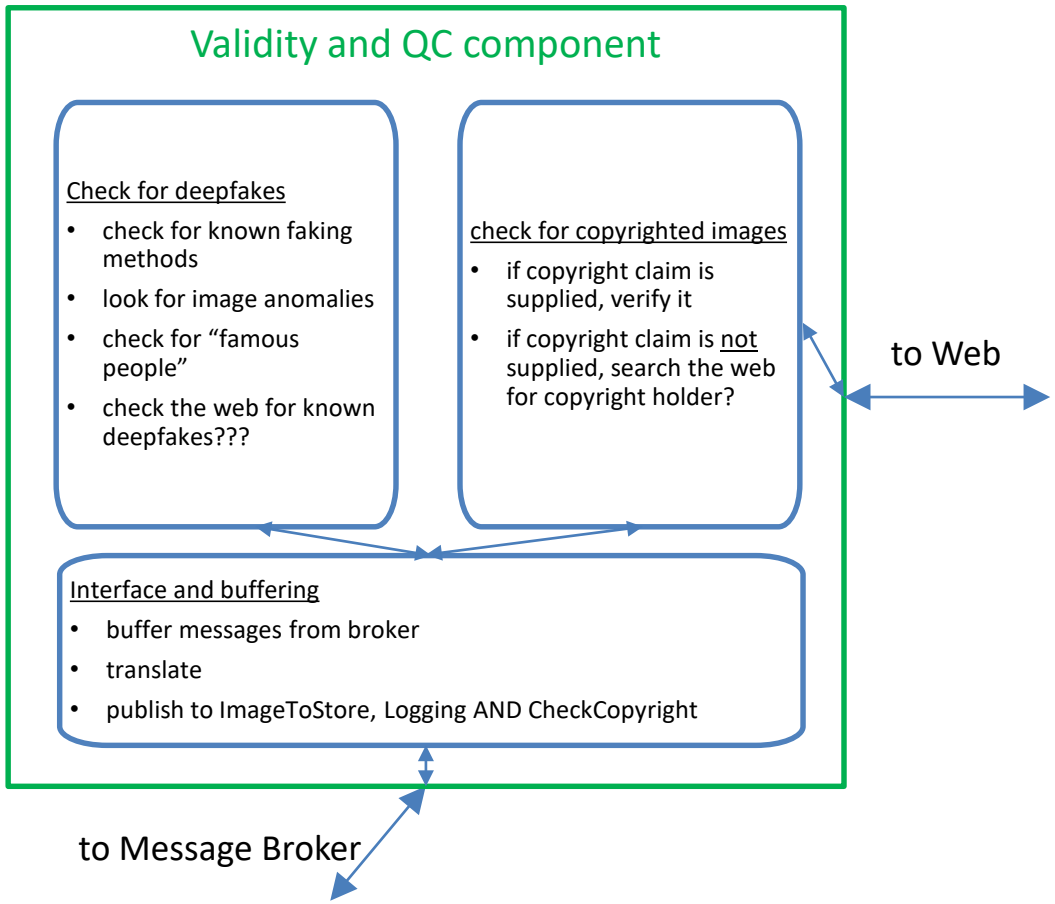


The Validity and QC component itself consists of other components – at a lower level

- I propose an Interface and Buffering component
  - handles interface with the broker
  - buffers requests
- Also, a set of Checking components
  - with common form and interface, as much as possible
  - Deepfake checking component
  - Copyright checking component
    - Needs to check the web
  - The need for others may arise, but we can easily add others

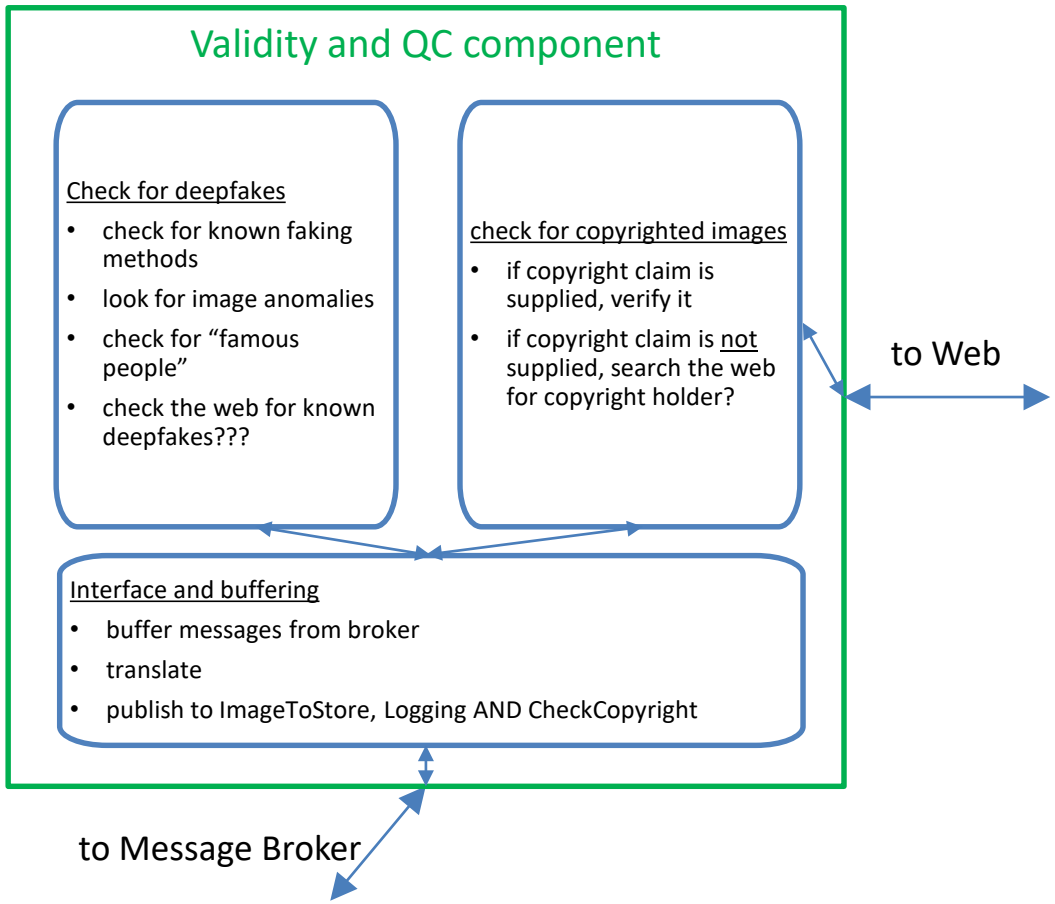


Architecting and designing a component is the place to look for *Design Patterns*



- The interface component can use the Adapter pattern to support the selected Message Broker's format
- The checking components could use Chain of Responsibility to modularize the work
- The image processing pieces should use the Strategy pattern, so different methods have the same interface
- The interface component might use the Mediator pattern to prevent the checking components from relying on each other
- <etc…>
- Don't overuse them!
  - Don't look for excuses to apply a pattern

At this stage we also look for opportunities to reuse existing code



- Do we have useful components or classes from an earlier project?
  - Shorter development time and more reliable code
- Are there open-source or commercially available classes or components that we can use?
- Does the same need appear in more than one place; can we write classes to serve several needs?
  - or can we modify the design to make it so?

Today's Objectives

Some thoughts on large software systems design

- decomposition
- WBS
- middleware

An example of an online photo retouching system

Architecture of a Component

- Component-based architecture
- Defining a component
- Designing a component