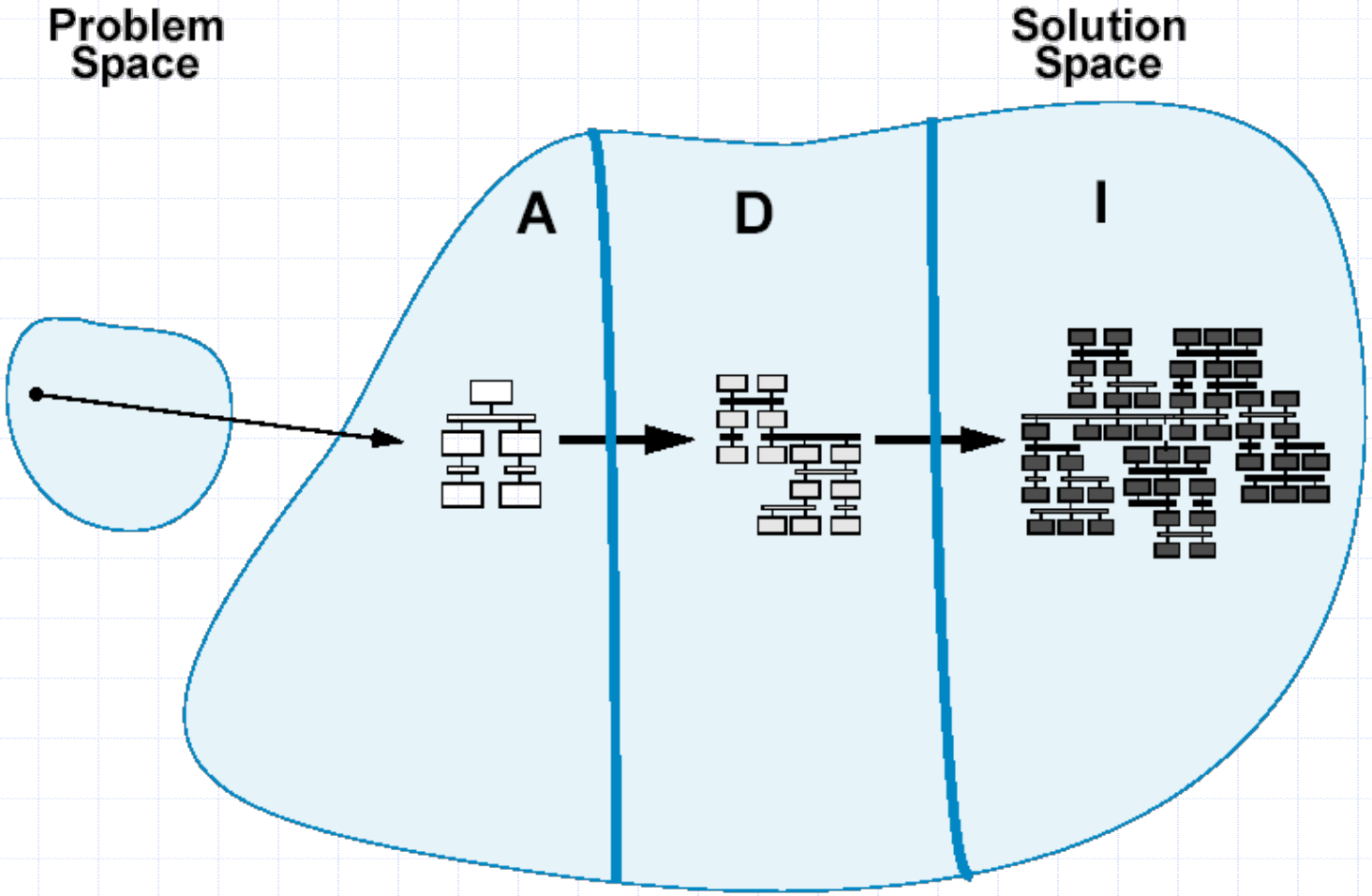
Modeling Software Architectures - ADLs, UML and Other Notations

Brian Mitchell

From Architecture to Design



The State-Of-Practice is NOT Improving for Modeling Software Design and Architecture

- Researchers and practitioners have different objectives…

Academic Focus	Practitioner Focus
Focus on analytic evolution of architecture models	Focus on a wide range of development issues
Individual models	Families of models
Rigorous modeling notations	Practicality over rigor
Powerful analysis techniques	Architecture as the “Big Picture” in development
Depth over breadth	Breadth over depth
Special-Purpose solutions	General Solutions

Revisiting Software Architecture

- The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components.
- Components needed to model a software architecture include:
  - Components, Connectors, Systems, Properties and Styles.

Software Architecture Concepts

- Components
  - The computational elements and data stores of the system
  - May have multiple interfaces, called ***ports***
  - ***Ports*** define a point of interaction between a component and its environment
- Connectors
  - Model interactions among components
  - Runtime perspective: connectors mediate the communication and coordination activities between components
  - Connectors may have interfaces that define the ***roles*** played by the participants in the interaction

Software Architecture Concepts

- Systems
  - Graphs of components and connectors
  - Tend to be hierarchical – components and connectors may represent ***subsystems*** that have their own internal architectures
  - ***Bindings*** map the interfaces of one level of a system to another
- Properties
  - Represent the non-structural information about the parts of an architecture description
  - Example: a connector can be a function call, or a network interaction
  - Properties can be attached to any architectural element

Software Architecture Concepts

- Style
  - An architectural style represents a family of related systems
  - Defines the design vocabulary (and constraints) for the components, connectors, ports, roles, bindings and properties.

So what are software architecture models?

- Architecture models capture the foundational design decisions about a system.
  - An architecture model is an **artifact** that captures these decisions
  - These models serve as documentation that can be consumed by various stakeholders

Modeling choices are important!

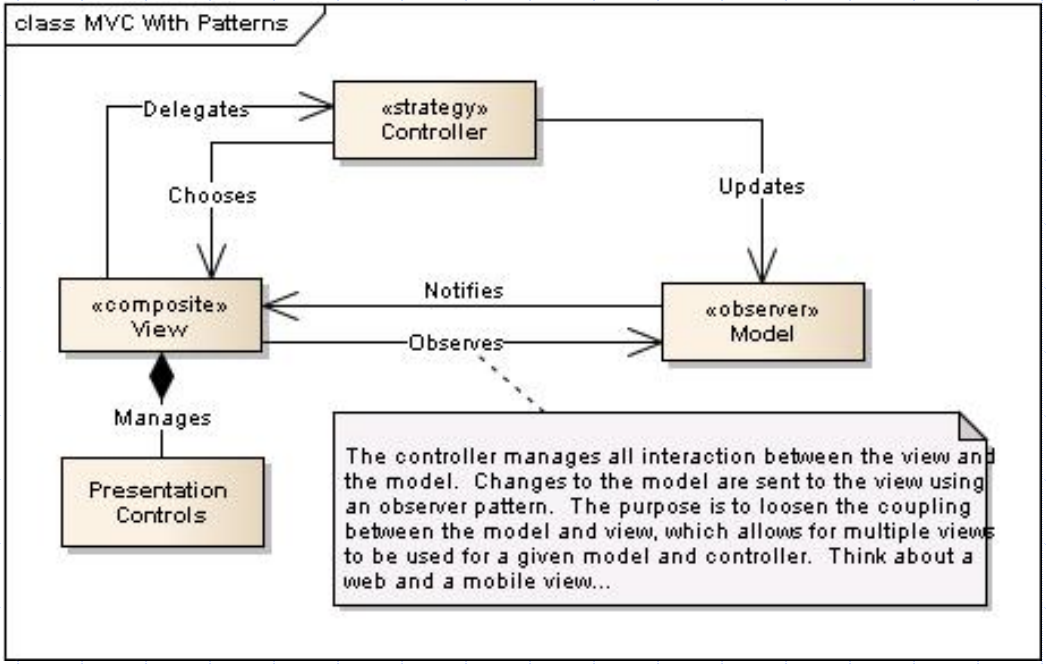
- The choice of what to model is important!
  - This activity takes time and costs money so picking what to model becomes a critical activity
- The choice of how detailed your models are are important!
  - What stakeholders are you trying to influence?
  - How much detail is needed to describe the architecture to the stakeholders?
  - Does the model describe the design, or is the model intended to influence a decision?
- The choice of what notation is used to document your architecture is important!
  - Should you use formal, or semi-formal notations

Modeling is about documenting design decisions!

- A **model** is an **artifact** that captures a number of different design decisions used to establish the overall system architecture
  - This activity takes time and costs money so picking what to model becomes a critical activity
- Key things to consider when modeling
  - What architectural decisions and concepts should be modeled,
  - At what level of detail, and
  - With how much rigor or formality

What do we document in a model?

- Components
- Connectors
- Interfaces (including constraints)
- Decisions and Rational
- Structural, Runtime and Behavioral Constraints

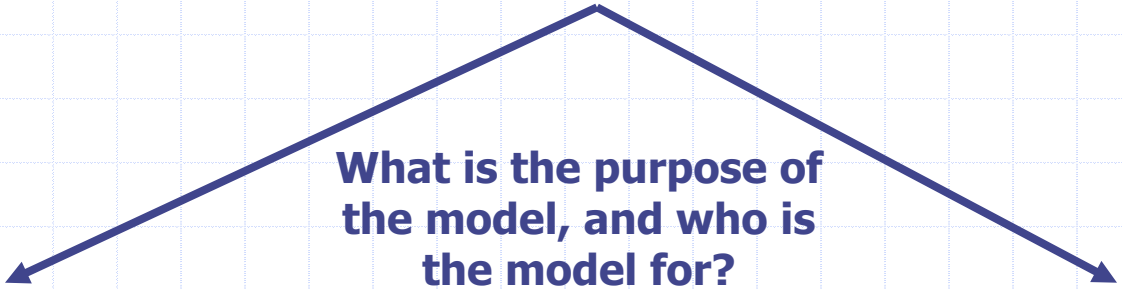


Remind you of the definition of software architecture?

Thinking about models…

To experiment
To clarify
To understand
To analyze
To evaluate

Reason for Modeling



What is the purpose of the model, and who is the model for?

What to Model

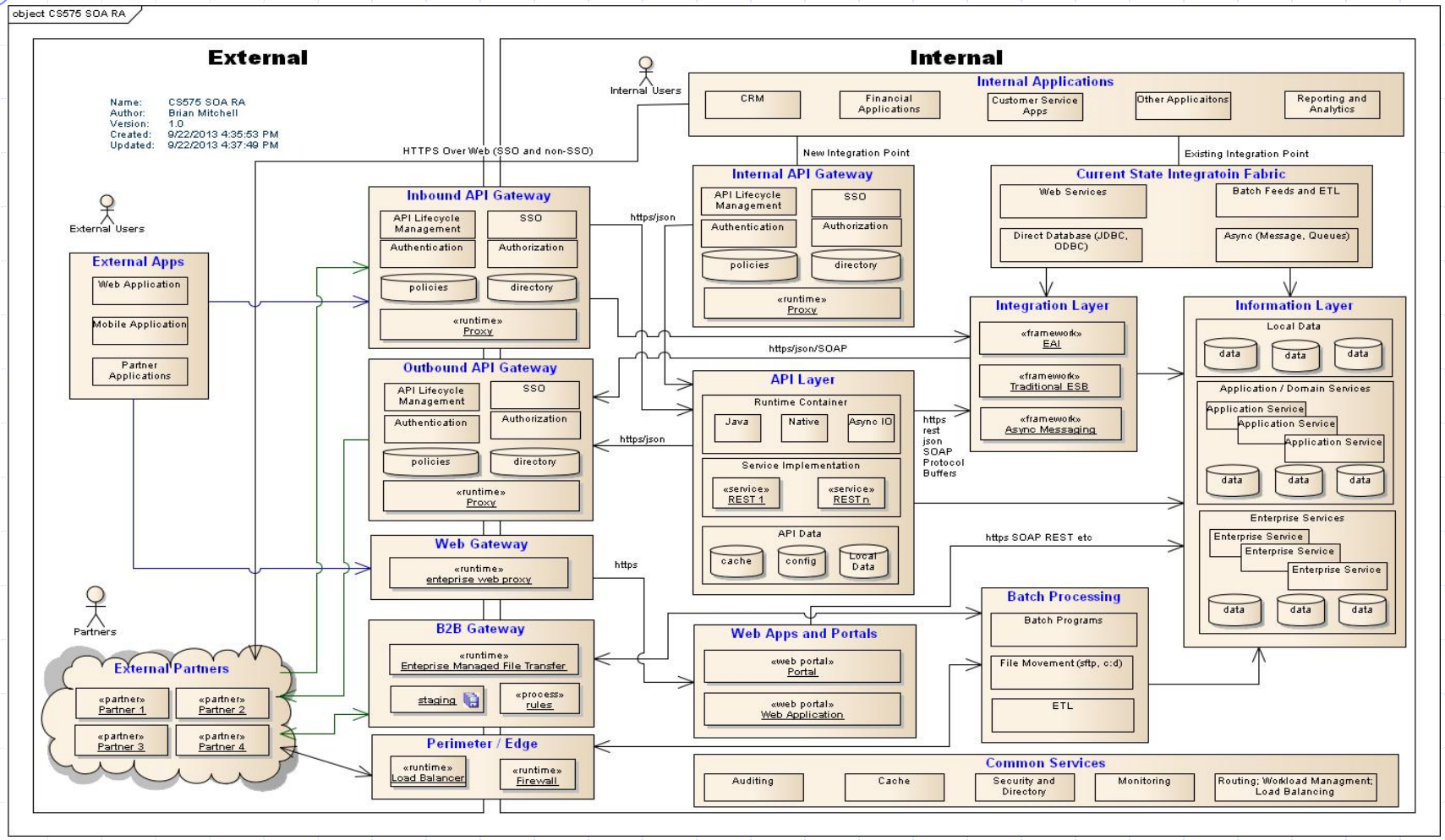
Structure
Transformations
State
Inputs and Outputs

How to Model

Textual
Graphical
Formal / Mathematical

Adapted From: https://www.cl.cam.ac.uk/teaching/1112/SWDesign/softwaredesign01.pdf

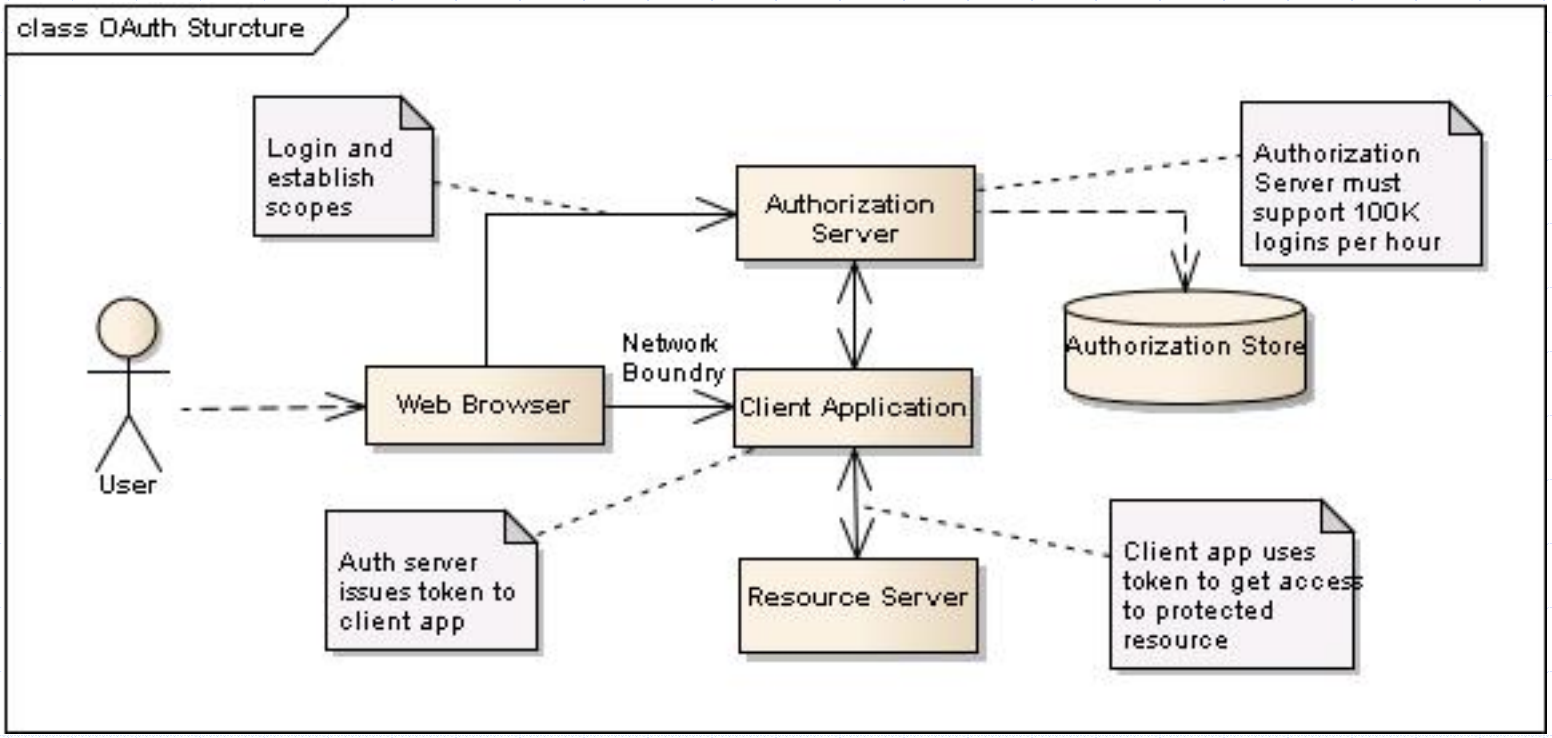
Example: What are some of the key aspects of this model?



Modeling the different views – Static and Dynamic

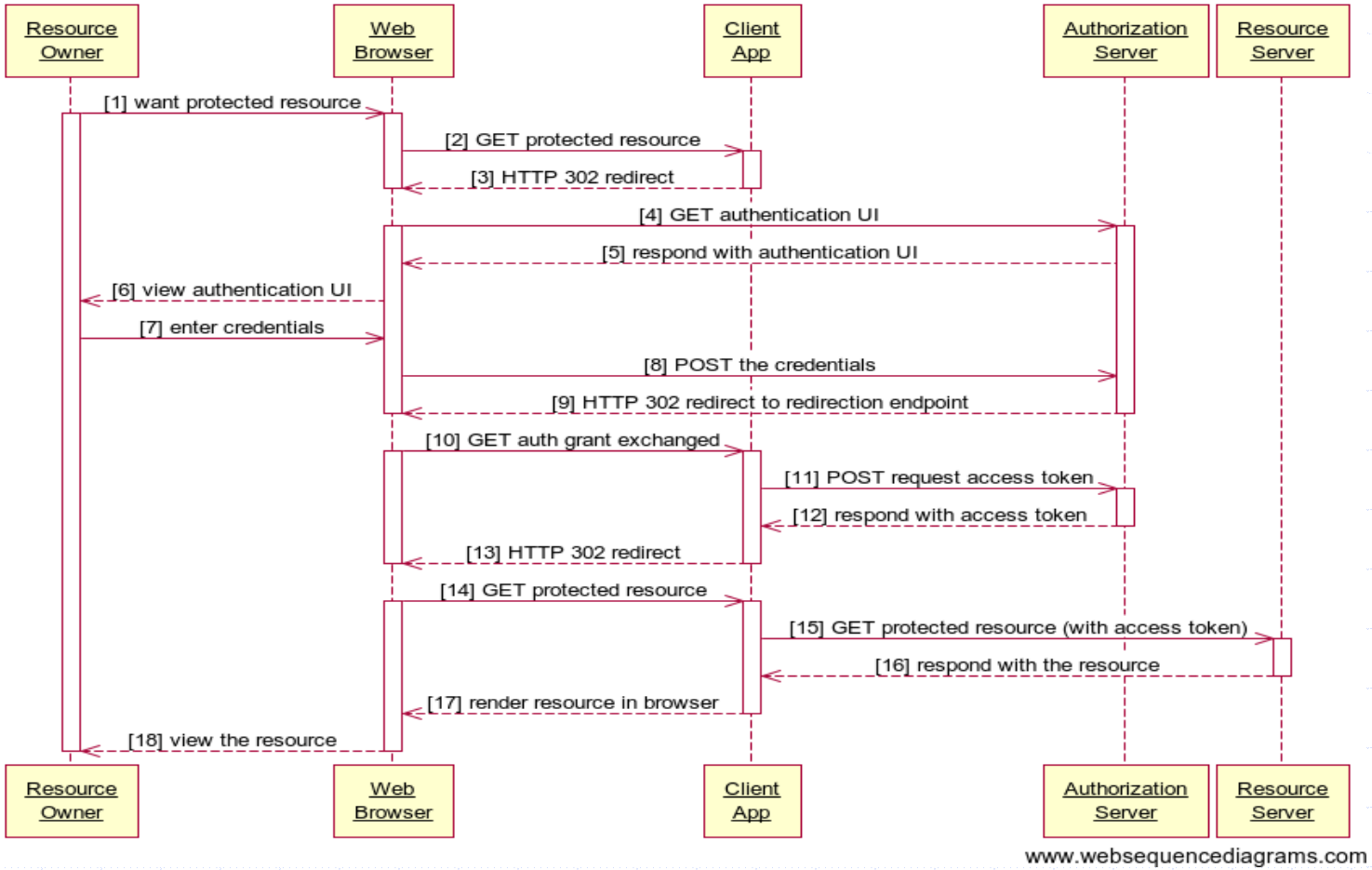
- Static models describe the structure of the system
  - These structures generally do not change over time
  - Capture the topology of the system components:
    - can be software component relations – uses/inherits
    - Can be runtime component relations – sends message to over HTTP
    - Can be deployment component relation – this software is deployed to this server…
- Dynamic models capture the behavior of the system during runtime
  - The important thing to capture is change is being managed over time
  - State of the components
  - Data flow over the connectors

Static Model View (Example: oAuth)



Is this view useful to discuss the components associated with managing the oAuth protocol (functional and non-functional)?

Dynamic Model View (Example: oAuth)



Is this view useful to discuss how the components implementing the oAuth protocol work together?

Possible problems using multiple views to model software architectures

- Modeling a realistic software architecture using a single view is not practical
  - Too complex
  - Different concerns that don't belong together – static, dynamic, deployment
- Key challenge is to ensure that all of the different views are consistent
  - Might be hard to find these issues
    - **Direct** – one diagram states two servers, another states three servers
    - **Indirect** – one view is at a higher level than another and the refinement introduced inconsistencies
    - **Structure vs Dynamic** – structures don't support dynamic requirements
    - **Functional vs Non-Functional** – The structure does not support the non-functional requirements

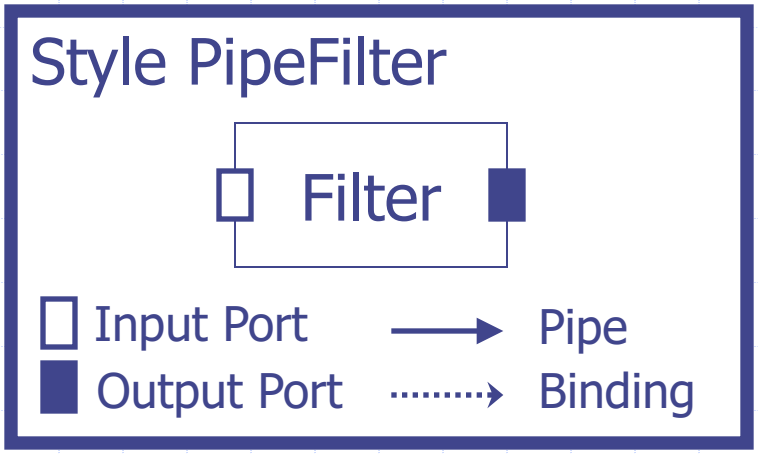
Describing Architectures

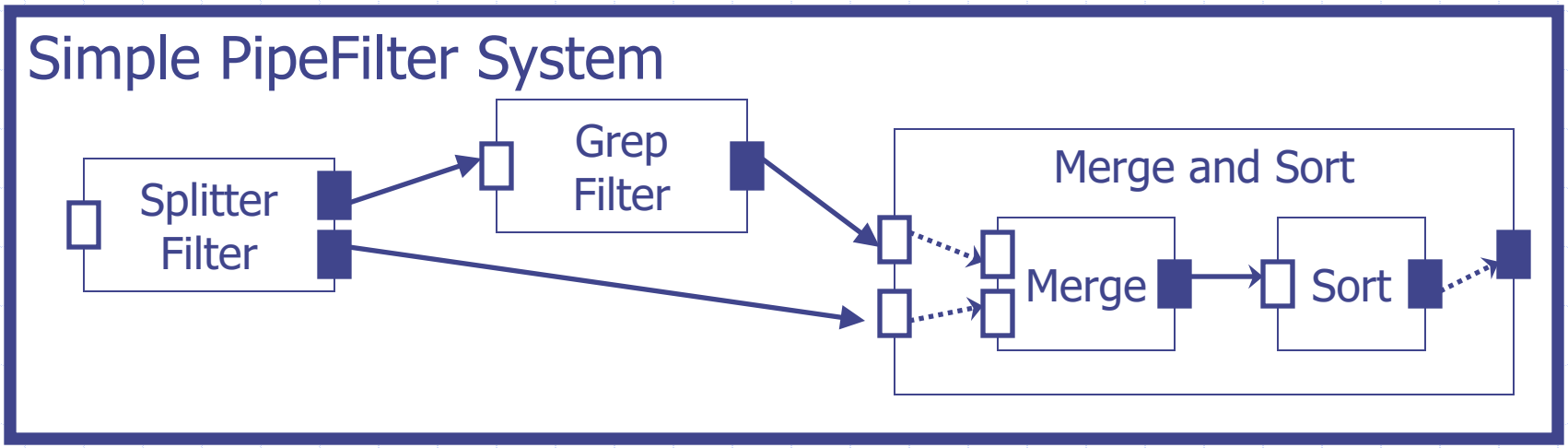
- Formal Approaches
  - Architectural Description Languages
  - Many to date
- Informal Approaches
  - UML Notation – Simple and known by a broader community
    - UML is missing architecture semantics
  - Lines and Boxes with natural English

Architecture Description Languages

- Formal languages to describe architectural components, their associated runtime views and semantics
- Some ADL's have associated tooling
- See David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek paper for examples

ADL – Example Pipe and Filter – Line and Box Drawing





ADL Example – Pipe and Filter

```
Family PipeFilter = {
  Port Type OutputPort;
  Port Type InputPort;
  Role Type Source;
  Role Type Sink;
  Component Type Filter;
  Connector Type Pipe = {
    Role src : Source;
    Role snk : Sink;
    Properties {
      latency : int;
      pipeProtocol: String = …;
    }
  };
};

System simple : PipeFilter = {
  Component Splitter : Filter = {
    Port pIn : InputPort = new InputPort;
    Port pOut1 : OutputPort = new OutputPort;
    Port pOut2 : OutputPort = new OutputPort;
    Properties { … }
  };
  Component Grep : Filter = {
    Port pIn : InputPort = new InputPort;
    Port pOut : OutputPort = new OutputPort;
  };
```

```
  Component MergeAndSort : Filter = {
    Port pIn1 : InputPort = new InputPort;
    Port pIn2 : InputPort = new InputPort;
    Port pOut : OutputPort = new OutputPort;
    Representation {
      System MergeAndSortRep : PipeFilter = {
        Component Merge : Filter = { … };
        Component Sort : Filter = { … };
        Connector MergeStream : Pipe = new Pipe;
        Attachments { … };
      }; /* end sub-system */
      Bindings {
        pIn1 to Merge.pIn1;
        pIn2 to Merge.pIn2;
        pOut to Sort.pOut;
      };
    };
  };
  Connector SplitStream1 : Pipe = new Pipe;
  Connector SplitStream2 : Pipe = new Pipe;
  Connector GrepStream : Pipe = new Pipe;
  Attachments {
    Splitter.pOut1 to SplitStream1.src;
    Grep.pIn to SplitStream1.snk;
    Grep.pOut to GrepStream.src;
    MergeAndSort.pIn1 to GrepStream.snk;
    Splitter.pOut2 to SplitStream2.src;
    MergeAndSort.pIn2 to SplitStream2.snk;
  };
}; /* end system */
```

The Problem with ADL's

- There are many of them that have been developed by the research community
- The desire for practitioners to learn another language is not there
- The process of modeling the architecture prior to modeling the design is new, and not widely applied
- ADL's are formal models, practitioners are not skilled in working with formal models

The Case for UML to Model Architecture

- UML is widely known
- There is wide commercial tool support for UML
- Research has shown that ADL formality can be “mapped” to UML
- UML is extensible via the UML meta-meta model
  - Changing UML to support modeling architecture components invalidates the desirability of using UML itself – if one needed to learn new UML features might as well just learn an ADL.

What UML can do (Out-of-the-Box)

- UML can model:
  - Classes and their declared attributes, operations and relationships
  - The possible states and behavior of the individual classes
  - Packages of classes and their dependencies
  - Example scenarios of system usage and the behavior of the overall system in the context of a usage scenario
  - Examples of object instances with actual attributes and relationships
  - The deployment and communication of software components on distributed hosts
- The goal of a UML model is to have HIGH FIDELITY where ***fidelity*** is the measure of how close the model corresponds to the eventual implementation

Extending UML

- The designers of UML realized that they couldn't create a language that effectively described the semantics of all possible situations.
  - UML has extensibility mechanisms that allow users to define custom extensions of the language, in order to accurately describe the semantics of specific information domains
- These extensions were intended to be applied to the design problems and not architectural concerns

Built-In Mechanisms to Extend UML

- *Tagged values*
  - Allows arbitrary information to be attached to model elements using keyword-value pairs called tags.
- *Stereotypes*
  - Allows sub-classification of model elements. Stereotypes can be used to introduce additional distinctions between model elements, that are not explicitly supported by the UML meta model.
- *Constraints*
  - Allows new semantic restrictions to be applied to elements. This makes it possible to linguistically specify additional constraints that should be obeyed by elements.
- *Profiles*
  - Profiles are pre-defined stereotypes, tagged values, constraints and icons to support modeling in a specific domain

The Design of UML

- UML is a semiformal graphical language
- UML is specified using:
  - A meta-model
  - Informal descriptive text
  - Constraints
- The UML metamodel is a UML model that specifies the abstract syntax of UML models
  - For example the UML metamodel specifies the ***class*** modeling element

UML: Models and Meta-Models

In meta-modeling terminology, both the model of a software system and the meta model, are layers in a meta-modeling architecture. A model of a system is considered an instance of the meta model.

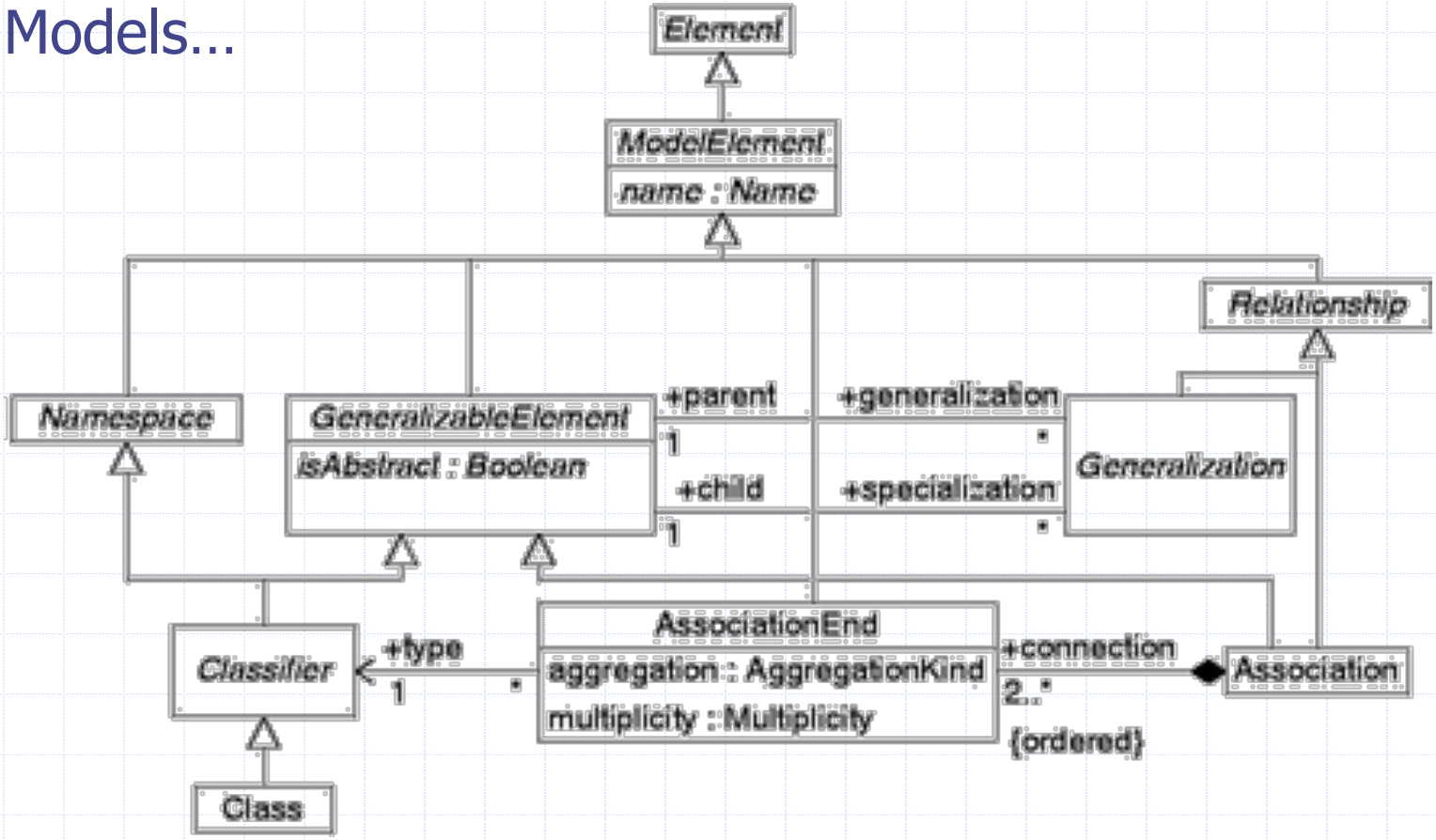
- The meta-modeling architecture for UML has a conceptual framework based on four meta-layers…

The UML Meta-Model Architectural Layers

Layer	Scope	Defines	Sample
Meta-Meta Model	general metamodeling infrastructure	Meta Model Specification Language	MetaClass, MetaOperation, MetaAttribute
Meta Model	UML Specification	Model Specification Language	Class, Component, Attribute, Operation
Model	Project Using UML	Domain Specification Language	List, Queue, Customer
User Object Data	Runtime System	Specific Domain	Customer [name=smith]

Part of the UML Meta-Model

- The UML Meta-Model Describes the Semantics of UML Models…



Constraining the UML meta-model

- A powerful UML extension mechanism to support architectural modeling can be achieved by constraining the way the meta model is used to construct system models
- For example, consider the built-in stereotype <<abstract>>, this introduces a constraint on how a class is instantiated, consider defining other custom stereotypes to constrain containment or interaction between components

Options for using UML as an ADL

- Use UML “as is”
- Constrain the UML meta-model using built in extension mechanisms
- Extend the UML meta-model (using the meta-meta model) to directly support needed software architecture concepts

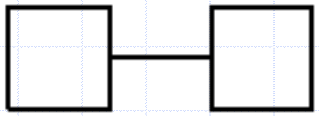
Strategy 1: Use UML "As Is"

- Simple Approach: Based on existing UML notation
- Architectural Models are understandable by users who already know UML
- Architectural Models can be manipulated by any UML-compliant tool
- The architectural concepts need to be simulated (overloaded) using existing UML notation

Strategy 1: Use UML "As Is"

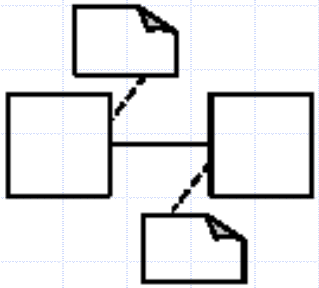
- Need to agree on notational conventions

generic (line-and-box)



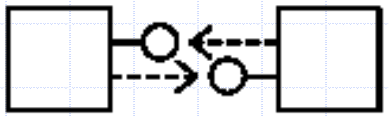
Option 1

ports as annotations

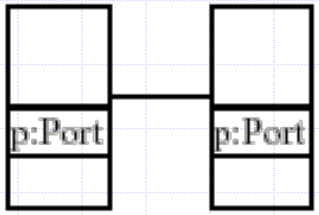


Option 2

ports as interfaces

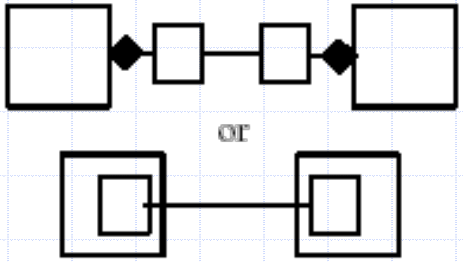


Option 3



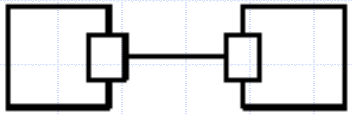
Option 4

ports as class attributes



Option 5

ports as classes



Option 6

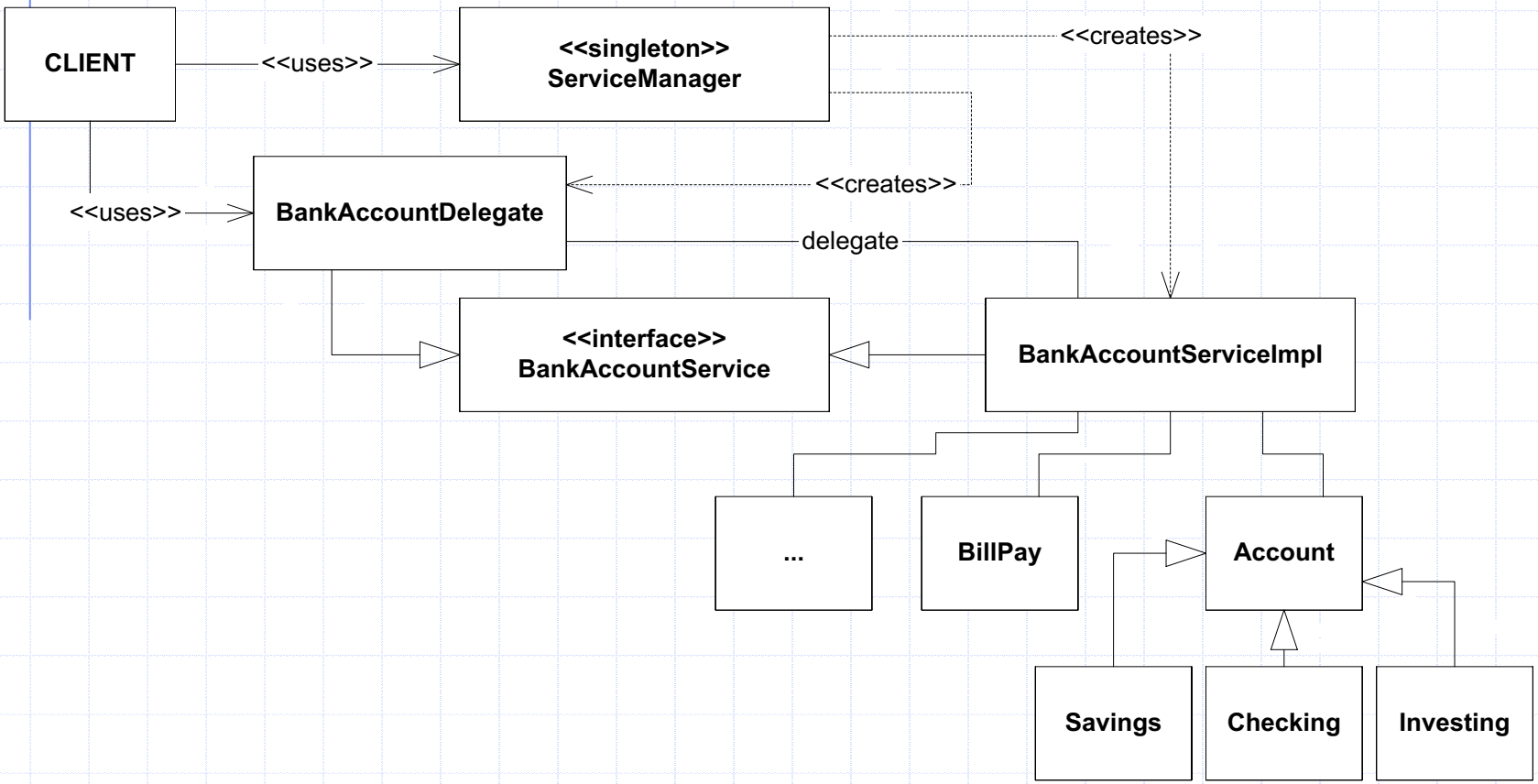
ports as classes (shown on boundary)

Strategy 1: Use UML "As Is"

- Simultaneous consideration of architecture rules and UML notational constructs
- Develop a UML domain model
- Develop an (informal) architectural diagram
- Map domain classes to architectural components
- Design class (component) interfaces
- Provide constructs for modeling connectors
  - Connectors add no functionality at the domain model level
- Model architectural structure in class, object, and collaboration diagrams
  - See Garlan 2001

Strategy 1: Use UML "As Is"

Service Locator, Business Delegate,...



Strategy 2: Constrain UML to Model Architectural Artifacts

- Use UML's built-in extension mechanisms on meta-classes
- Allows automated conformance checking
- Select a meta-class semantically close to an ADL construct
- Define a stereotype and apply it to meta-class instances
- Class semantics are constrained to that of the ADL

Strategy 2: Constrain UML to Model Architectural Artifacts

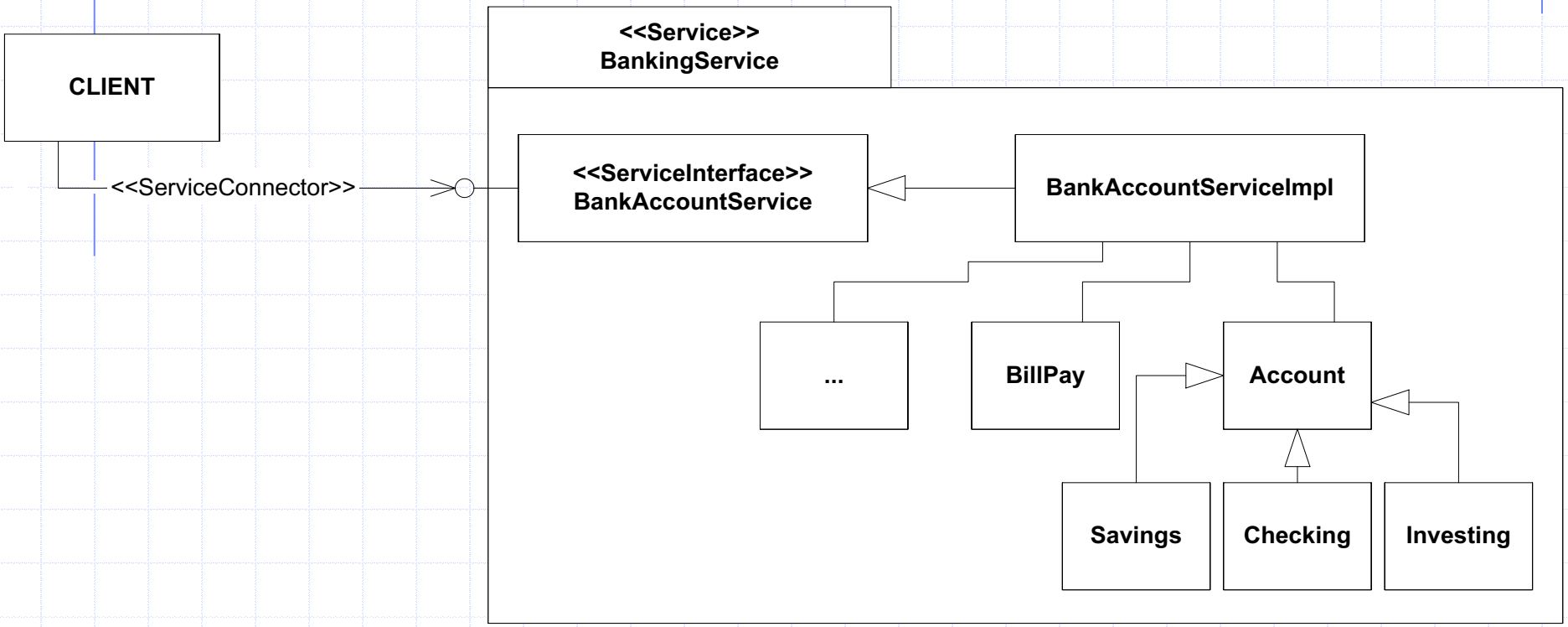
- A suggested process:
  - Identify UML constructs that map closest to an architectural construct:
    - Components, Connectors, Systems, Properties and Styles
  - Use stereotypes to constrain the UML entities
  - Describe semantics using UML statecharts and Object Constraint Language

OCL – What is it, Why do we need it…

- Full Reference: http://umlcenter.visual-paradigm.com/umlresources/obje_11.pdf
- OCL is a formal language to specify side-effect free constraints
- OCL used to specify well-formedness rules of the UML metamodel
- Why OCL: Graphical languages like UML cannot produce an unambiguous specification
  - Need to describe additional constraints about objects in the model
- OCL can be used to develop architecture-appropriate constraints that can used by OCL-compliant tooling

Strategy 2: Constrain UML to Model Architectural Artifacts

Define Custom Stereotypes



Strategy 3: Extend UML

- Use UML's meta metamodel to extend UML
  - Introduce *explicit* architectural constructs and in UML
- Introduce additional notations for modeling architectural semantics
- Follow an approach similar to strategy #1 to model specific architectures
- Follow an approach similar to strategy #2 to model specific architectural styles

Comparing the Approaches

- "Straight" UML
  - understandable architectures
  - manipulable by standard tools
  - architectural constraint violations
- "Constrained" UML
  - ensures architectural constraints
  - requires complete style specifications
  - requires OCL-compliant tools
- "Extended" UML
  - provides "native" support for architectures
  - requires backward tool compatibility
  - may result in incompatible UML versions
  - must be careful not to create another "box and line" notation, invalidating the motivation to use UML in the first place

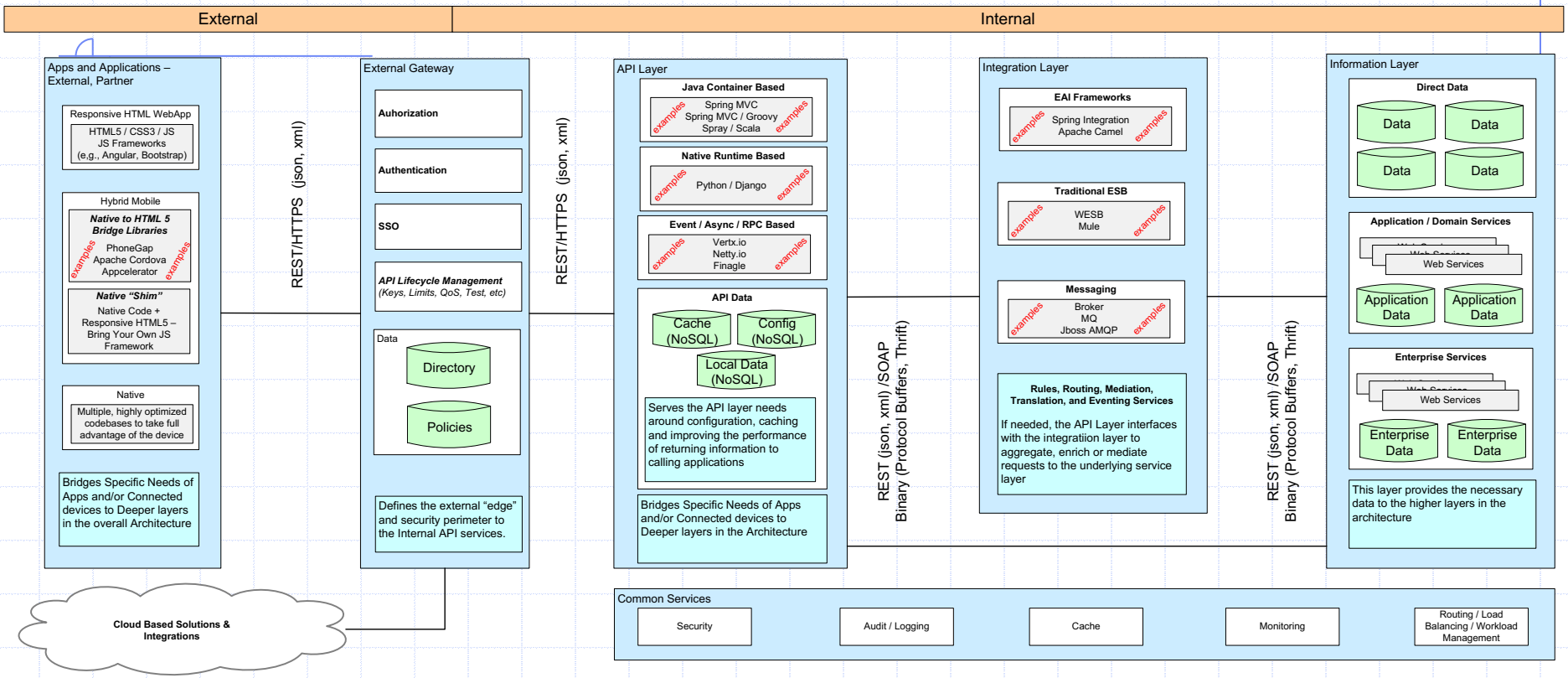
Example: Revisiting ADL's for Architectural Modeling

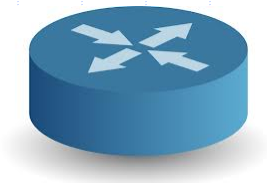
- Recall that "strategy 2" involved constraining UML to model architectural artifacts
- Although useful for design, we have also demonstrated that certain types of ADL's can be used for architectural maintenance.

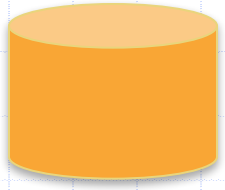
Using informal lines and boxes to model architecture

- In this approach a diagraming tool such as Microsoft Visio or Omnigraffle is used to document the architecture views
- Tends to create the "best looking" views given that there is no limitation to the artwork that can be applied to the components and connectors.
- Probably the most popular approach to document architectures
- But there are challenges:
  - There is a lack of rigor over the component and connector vocabulary
  - They tend to be pictures, and difficult to get value over managing the views in a model repository – for example – there is limited opportunity to share metadata between diagrams
  - Hard to deal with versioning and change – "Is this the latest view?"
  - Impossible to detect inconsistencies between models

Example: Line and Box Drawing for SOA Architecture







Comparing the approaches

Approach	Pros	Cons
ADL	• Most formal of all of the approaches • Able to precisely define architecture components, connectors and constraints	• Not visual • Too complex for use outside of the academic community • No wide standard • Almost no activity in the research community these days, and not used in industry to the best of my knowledge
UML	• Supports the basic constructs to support architecture concerns • Can be integrated into a model repository promoting reuse • Good support for maintaining relationships between models	• UML is really designed for OOP where design can be mapped to code • Quasi-formalism may lead to inconsistency between different designers • Attempts to add architecture semantics to UML are challenging
Lines and Boxes	• Just drawings – can represent any architecture concern • Easiest to create models that can be digested by the widest number of stakeholders • Produces the nicest looking views	• Just artwork, can be interpreted differently by different people • Typically just managed as files (not versioned in a repository) which can lead to version management problems. • No direct support for maintaining relationships between models

And now a shameless plug into some interesting research on architecture recovery and architecture style repair

ISF: Interconnection Style Formalism

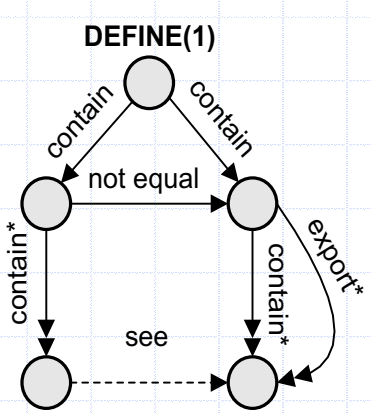
- ISF is a visual language used to define *Interconnection Styles*
- Interconnection Styles are specified in ISF using a series of rules that define structural and semantic properties of architectural relations

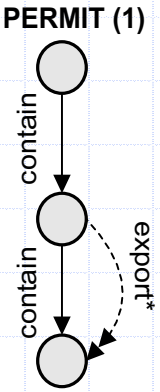
ISF Rules

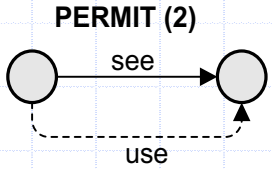
- ISF allows for the definition of 2 types of rules…
  - *Permission rules*, which define the set of well-formed configurations of sub-systems, modules, usage and architectural relations that adhere to a specific style.
  - *Definition rules,* which are used to define new relations based on patterns of components and relations.

ISF Style Examples – Export and Broad Export

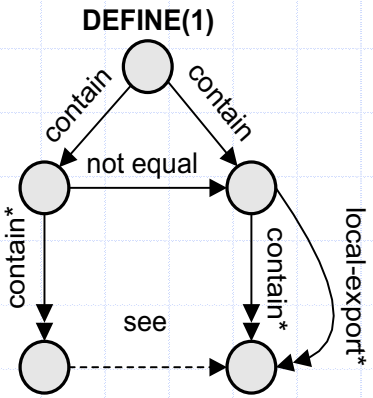
Export Style

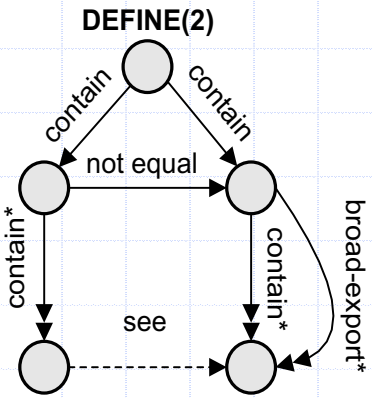


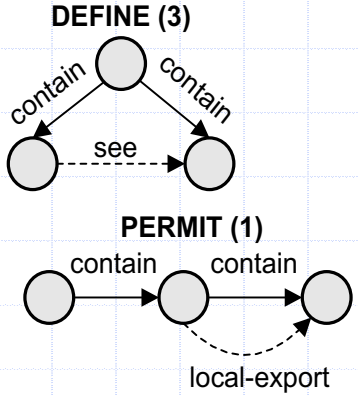


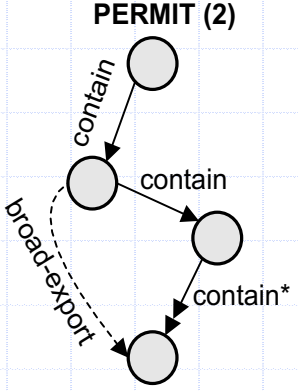


Broad Export Style

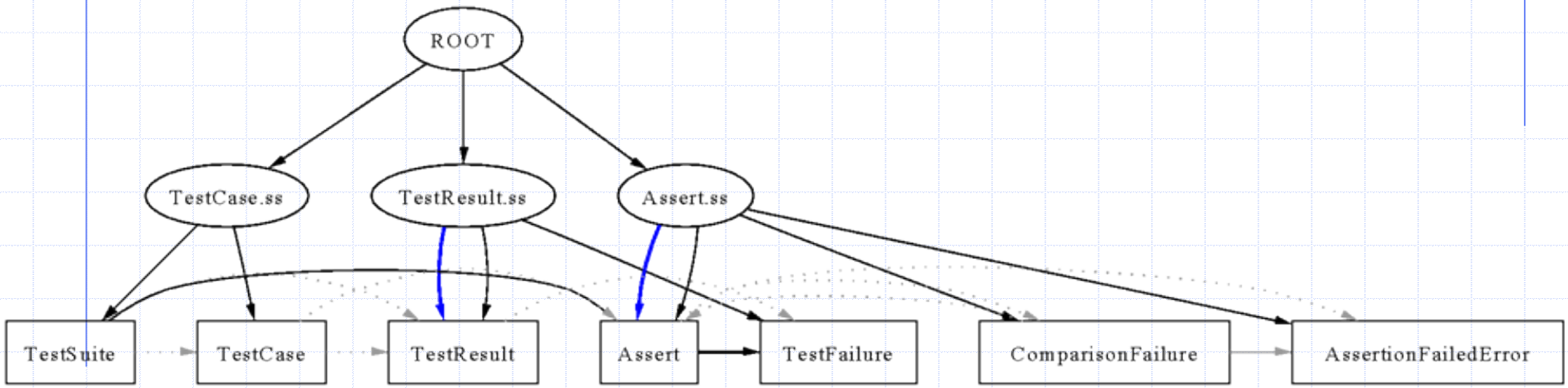








A Small Example



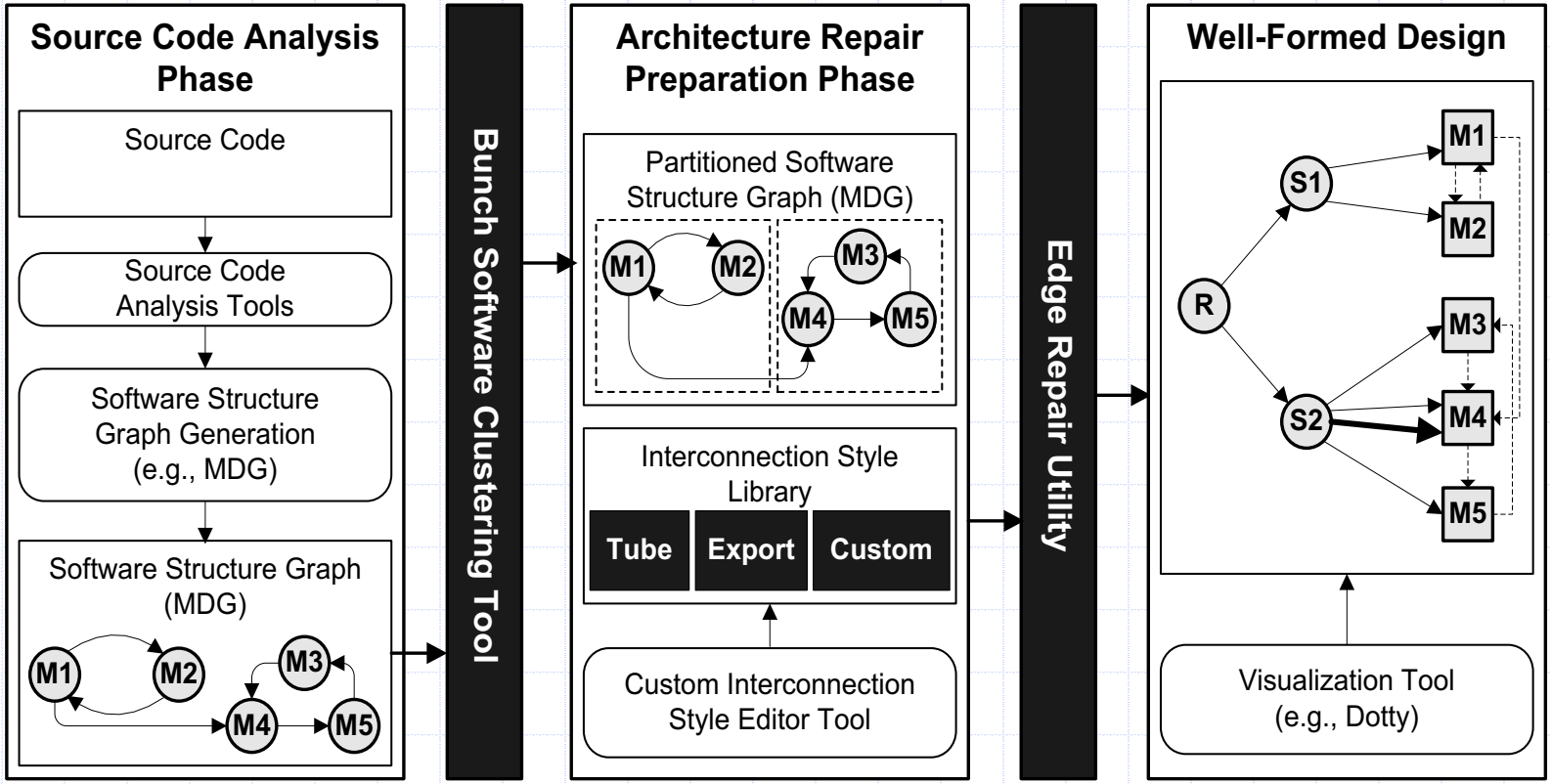
Relation Types

Use [from source code]

Contain [from clustering process]

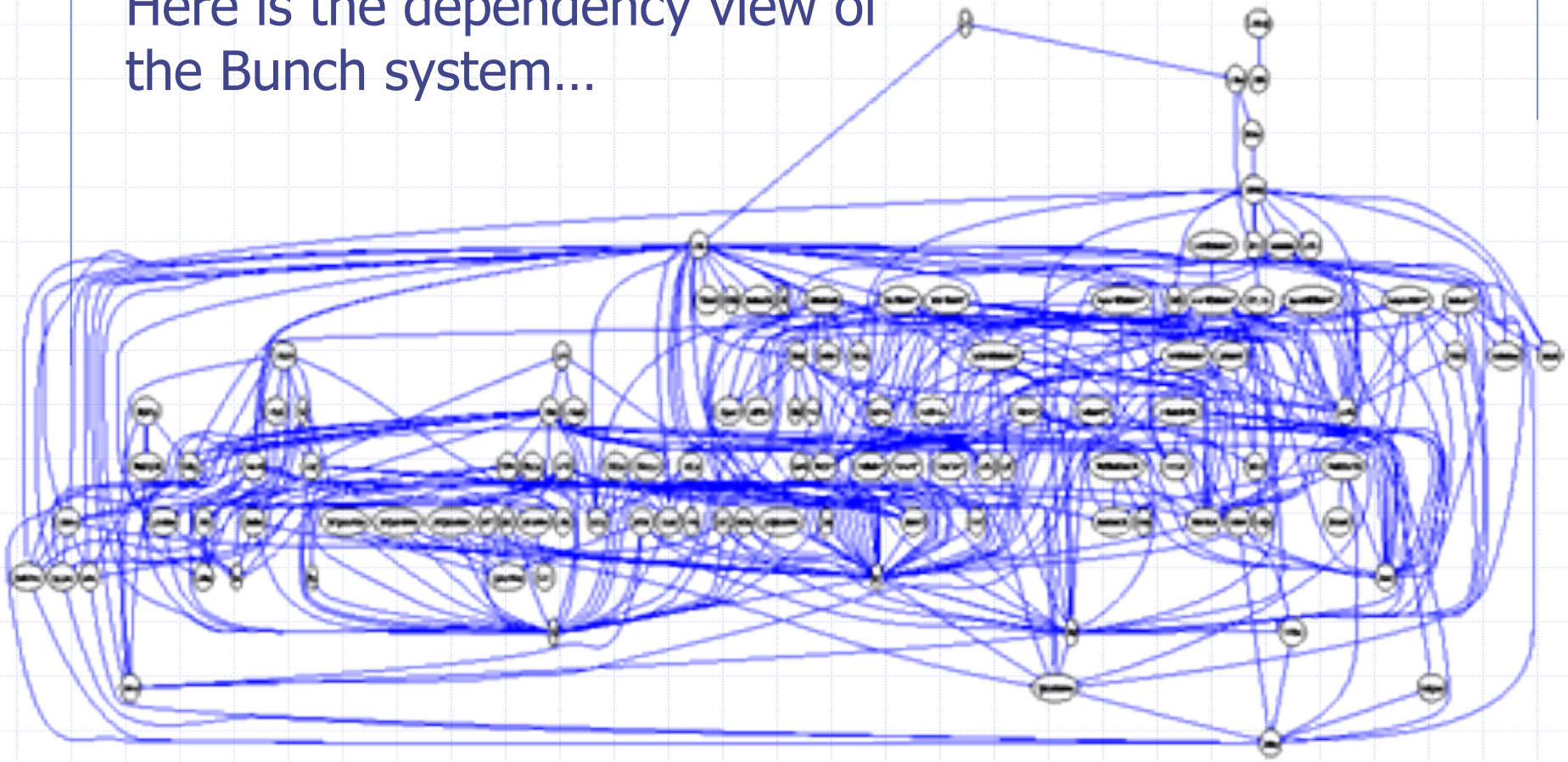
Export [from architectural recovery]

Example: Using ISF for Architectural Maintenance [from Mitchell,Mancoridis 2004]



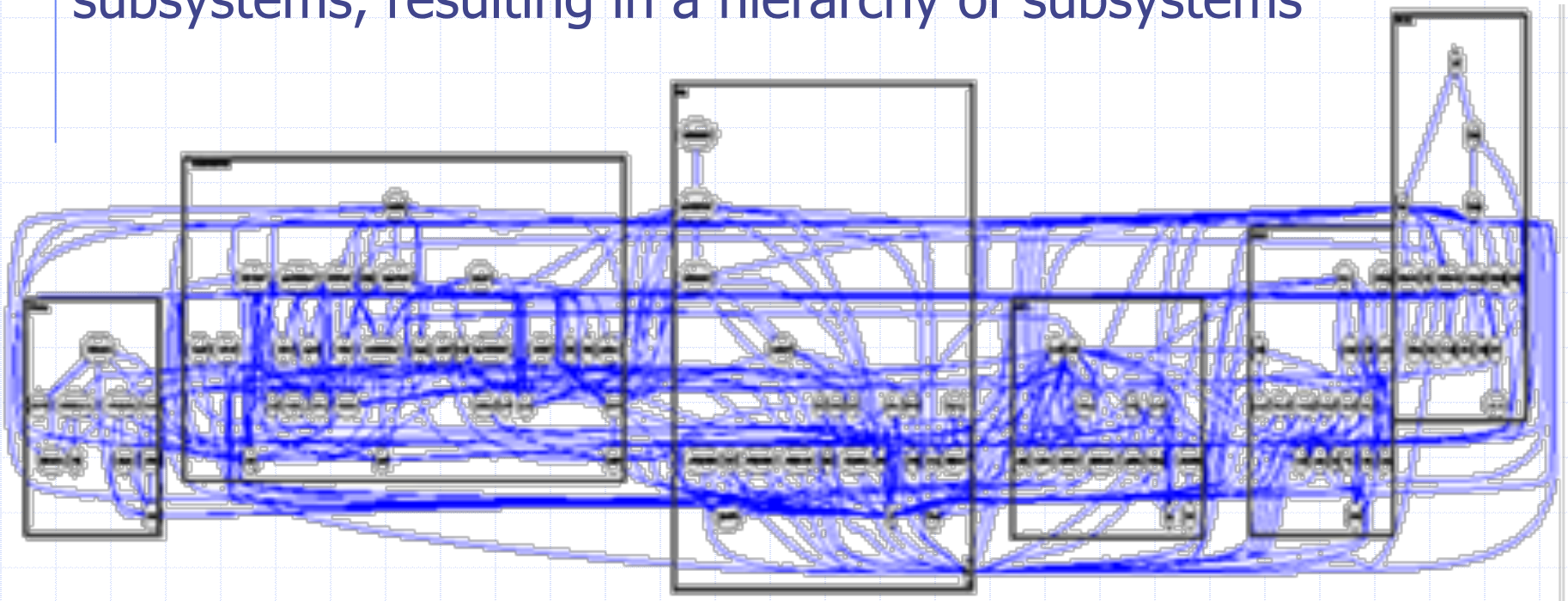
Example: Using ISF for Architectural Maintenance

Here is the dependency view of the Bunch system…



Example: Using ISF for Architectural Maintenance

Here is the automatically produced decomposition of Bunch's structure showing 6 abstract subsystems. Each of these Subsystems are comprised of one or more additional subsystems, resulting in a hierarchy of subsystems

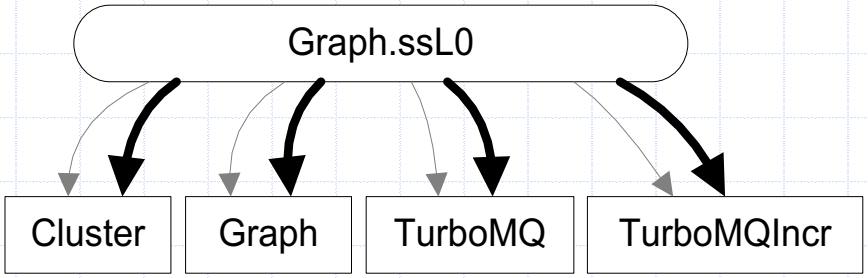


An Example Showing A “Reengineering” opportunity

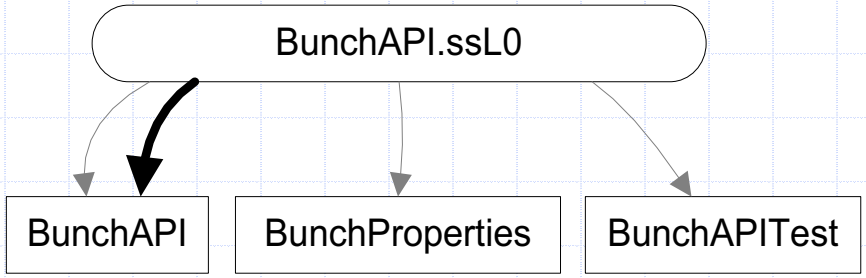
BAD
High Visibility

GOOD
High Encapsulation

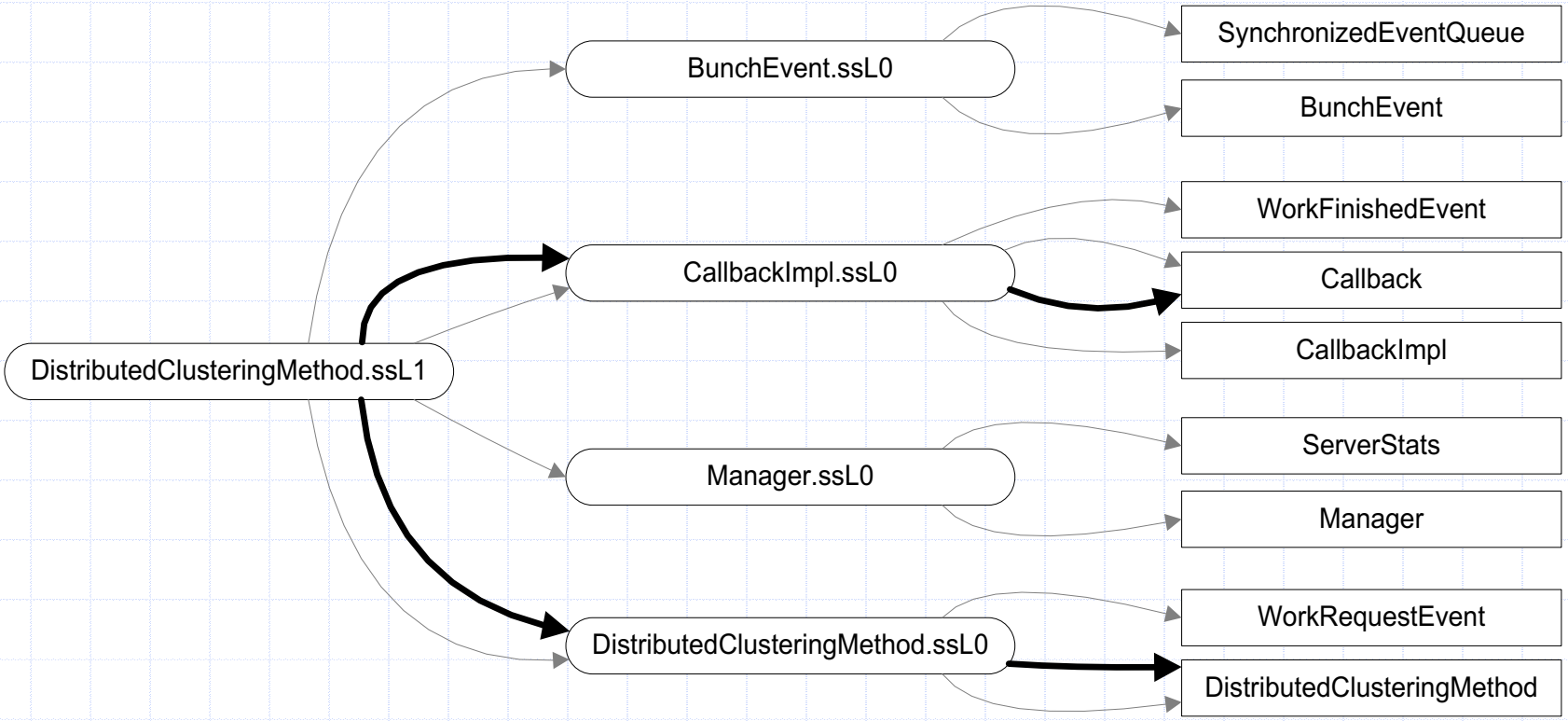
The Graph Subsystem



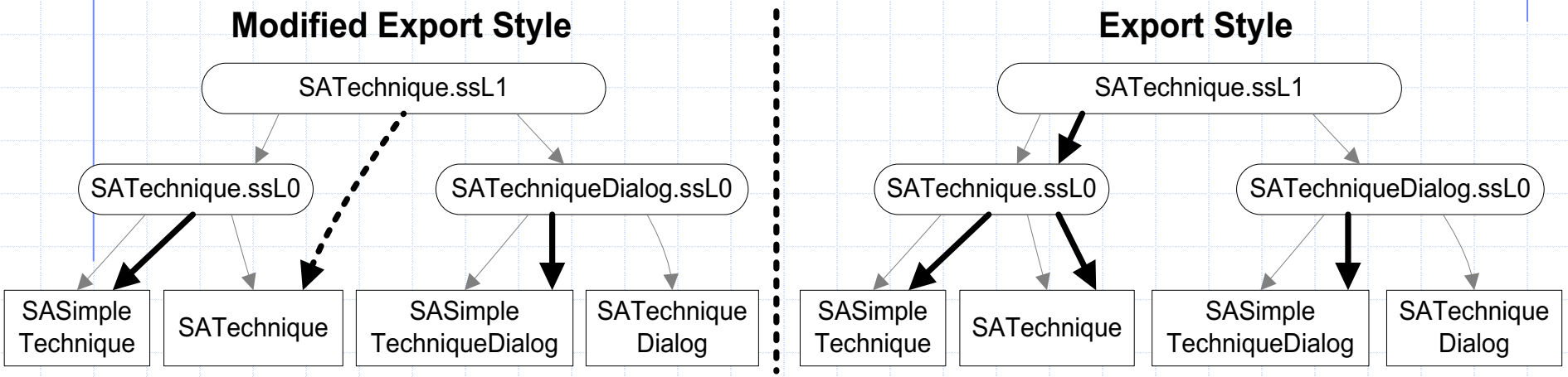
The BunchAPI Subsystem



What we can learn about the Export Style



Example: Export versus Modified Export Style



References

- A framework for the UML meta model – online at: http://www.ii.uib.no/~rolfwr/thesisdoc/main1.html
- Nenad Medvidovic, Class Notes: http://sunset.usc.edu/classes/cs578_2004/April6b.pdf
- David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek, "Reconciling the Needs of Architectural Description with Object-Modeling Notations", Science of Computer Programming Journal, 2001.
- Nenad Medvidovic, David Rosenblum, David Redmiles, Jason Robberts, "Modeling Software Architectures in the Unified Modeling Language", ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 1, Jan 2002.
- Brian S. Mitchell, Spiros Mancoridis, and Martin Traverso, "Using Interconnection Style Rules to Infer Software Architecture Relations", Proceedings of the 2004 Genetic and Evolutionary Computational Conference (GECCO), June 2004.
- R. Taylor, N. Medvidovic and E. M. Dashofy, "Software Architecture: Foundations, Theory and Practice", Wiley, January 2009.