MCS 680 Software Design

by

Spiros Mancoridis

smancori@mcs.drexel.edu

http://www.mcs.drexel.edu/~smancori

Course Overview

-
- Background
- Research Contributions:
  - Architectural Scoping Formalism
  - Scoping Rules for Programming
  - Scoping Rules for Software Architecture
  - Tool Integration Framework
- Future Work

Software Architecture Topics

- Terminology and Motivation
- Intuition About Architecture:
  - Hardware Architecture
  - Network Architecture
  - Building Architecture
- Architectural Styles of Software Systems:
  - Pipe and Filter
  - Object-Oriented
  - Implicit Invocation
  - Client-Server
  - Layered
  - Repository
  - Interpreter
  - Process-Control
- Case Study: Compiler Architecture
- The Future of Software Architecture

Terminology and Motivation

Abstraction

One characterization of progress in software development has been the regular **increase in abstraction levels** – or the conceptual size of software designer's building blocks.

Early 1950s: Software was written in machine language:

- programmers placed instructions and data individually and explicitly in the computer's memory,
- insertion of a new instruction in a program might require hand checking the entire program to update references to data and instructions.

Material drawn from [Garlan and Shaw, 1993, Shaw and Garlan, 1996, Shaw, 1989, Perry and Wolf, 1992].

Assemblers

Some major problems associated with machine code programming were solved by introducing a new level of abstraction between the program and the machine:

- **Symbolic Assemblers:** Names used for operation codes and memory addresses. $\Rightarrow$ Memory layout and update of references are automated.
- **Macro Processors:** Allow a single symbol to stand for a commonly used sequence of instructions.

Programming Languages

Late 1950s: The emerging of the first high-level programming languages. Well understood patterns are created from notations that are more like mathematics than machine code.

- evaluation of arithmetics expressions,
- procedure invocation,
- loops and conditionals

FORTRAN becomes the first widely used programming language.

Algol and its successors followed with higher-levels of abstraction for representing data (types).

Abstract Data Types

Late 1960s and 1970s: Programmers shared an intuition that good data structure design will ease the development of a program.

This intuition was converted into theories of modularization and information hiding.

- Data and related code are encapsulated into modules.
- Interfaces to modules are made explicit.

Various programming languages (*e.g.*, Modula, Ada, Euclid) and Module Interconnection Languages (*e.g.*, MIL75, Intercol) emerge with implementations of this theory.

Software Architecture

As the size and complexity of software systems increases, the design problem goes beyond algorithms and data structures.

Designing and specifying the overall system structure (**Software Architecture**) emerges as a new kind of problem with issues such as:

- organization and global control structure
- protocols of communication, synchronization, and data access
- assignment of functionality to design elements
- physical distribution
- selection among design alternatives

State of Practice

There is not currently a well-defined terminology or notation to characterize architectural structures.

However, good software engineers make common use of architectural principles when designing complex software.

These principles represent rules of thumb or idiomatic patterns that have emerged informally over time. Others are more carefully documented as industry standards.

Descriptions of Architectures

"Camelot is based on the *client-server model* and uses remote procedure calls both locally and remotely to provide communication among applications and servers."

"*Abstraction layering* and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a *client-server model* for the structuring of applications."

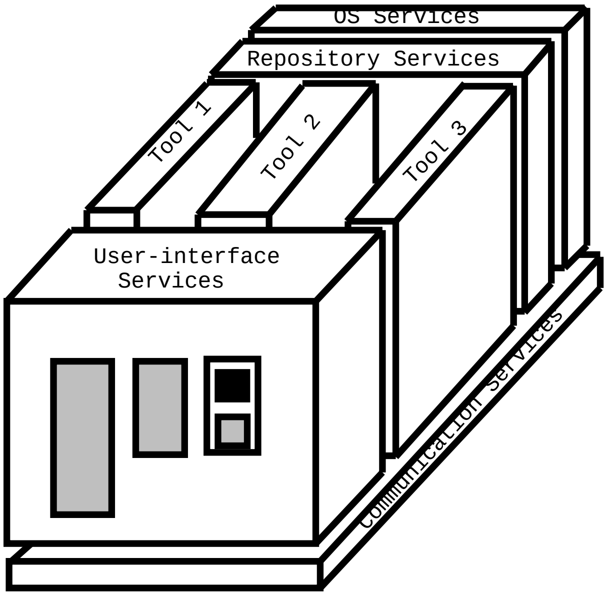
Descriptions of Architectures (Cont.)

"We have chosen a *distributed*, *object-oriented* approach to managing information."

"The easiest way to make a canonical sequential compiler into a concurrent compiler is to *pipeline* the execution of the compiler phases over a number of processors. A more effective way is to split the source code into many segments, which are concurrently processed through the various phases of compilation (by multiple compiler processes) before a final, merging pass recombines the object code into a single program."

Some Standard Architectures

- *ISO/OSI Reference Model* is a layered network architecture.
- *X Window System* is a distributed windowed user interface architecture based on event triggering and callbacks.
- *NIST/ECMA Reference Model* is a generic software engineering environment architecture based on layered communication substrates.



Intuition About Architecture

Intuition About Architecture

It is interesting that we have so few *named* software architectures. This is not because there are so few architectures, but so many.

We next look at several architectural disciplines in order to develop an intuition about software architecture. Specifically, we look at:

- Hardware Architecture
- Network Architecture
- Building Architecture

Hardware Architecture

RISC machines emphasize the instruction set as an important feature.

Pipelined and *multi-processor* machines emphasize the configuration of architectural pieces of the hardware.

Differences between HW and SW architectures:

- relatively (to software) small number of design elements.
- scale is achieved by replication of design elements.

Similarities: we often configure software architectures in ways analogous to hardware architectures. (*e.g.*, we create multi-process software and use pipelined processing).

Network Architecture

Networked architectures are achieved by abstracting the design elements of a network into nodes and connections.

Topology is the most emphasized aspect:

- Star networks
- Ring networks
- Manhattan Street networks

Unlike software architectures, in network architectures only few topologies are of interest.

Building Architecture

- **Multiple Views:** skeleton frames, detailed views of electrical wiring, etc.
- **Architectural Styles:** Classical, romanesque, etc.
- **Materials:** One does not build a skyscraper using wooden posts and beams.

Architectural Styles of Software Systems

Architectural Style of Software Systems

An *Architectural Style* defines a family of systems in terms of a pattern of structural organization. It determines the **vocabulary** of components and connectors that can be used in instances of that style, together with a set of **constraints** on how they can be combined.

Architectural constraints:

- may have to do with the topology of the descriptions (*e.g.*, no cycles).
- may have to do with execution semantics (*e.g.*, processes execute in parallel).

Determining an Architectural Style

We can understand what a style is by answering the following questions:

- What is the structural pattern? (*i.e.*, components, connectors, constraints)
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are some common examples of its use?
- What are the advantages and disadvantages of using that style?
- What are some of the common specializations of that style?

Describing an Architectural Style

The architecture of a specific system is a collection of computational components together with a description of the interactions between these components (connectors).

- software architectures are represented as graphs where:
  - Nodes represent components:
    - procedures
    - modules
    - processes
    - tools
    - databases
  - Edges represent connectors:
    - procedure calls
    - event broadcasts
    - database queries
    - pipes

Pipe and Filter Architectural Style

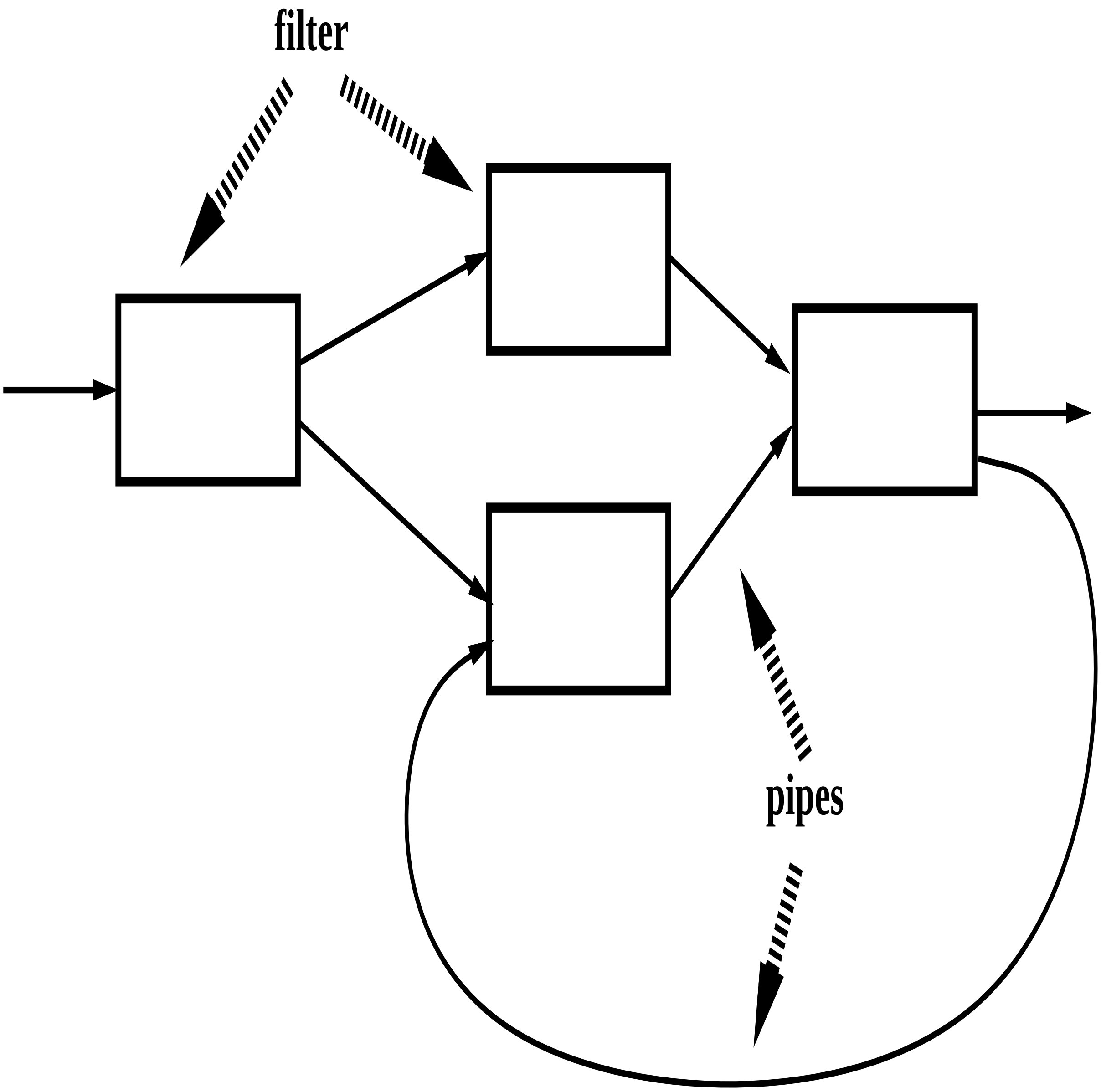
Suitable for applications that require a defined series of independent computations to be performed on ordered data.

A component reads streams of data on its inputs and produces streams of data on its outputs.

Components, called *filters*, apply local transformations to their input streams and often do their computing incrementally so that output begins before all input is consumed.

Connectors, called *pipes*, serve as conduits for the streams, transmitting outputs of one filter to inputs of another.

Pipe and Filter Architectural Style



Pipe and Filter Invariants

- Filters **do not share state** with other filters.
- Filters **do not know the identity** of their upstream or downstream filters.
- The **correctness** of the output of a pipe and filter network should not depend on the **order** in which their filters perform their incremental processing.

Pipe and Filter Specializations

- **Pipelines:** Restricts topologies to linear sequences of filters.
- **Batch Sequential:** A degenerate case of a pipeline architecture where each filter processes all of its input data before producing any output.

Pipe and Filter Examples

- **Unix Shell Scripts:** Provides a notation for connecting Unix processes via pipes.

  cat file | grep Erroll | wc -l

- **Traditional Compilers:** Compilation phases are pipelined, though the phases are not always incremental. The phases in the pipeline include:

  lexical analysis → parsing →
  semantic analysis → code generation

Pipe and Filter Advantages

- **Easy to understand** the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters.
- They **support reuse**, since any two filters can be hooked together, provided they agree on the data that is being transmitted between them.
- Systems can be **easily maintained and enhanced**, since new filters can be added to existing systems and old filters can be replaced by improved ones.
- They permit certain kinds of **specialized analysis**, such as throughput and deadlock analysis.
- The naturally **support concurrent execution**.

Pipe and Filter Disadvantages

- Not good for handling **interactive systems**, because of their transformational character.
- Excessive parsing and unparsing leads to loss of **performance** and increased **complexity** in writing the filters themselves.

Object-Oriented Style

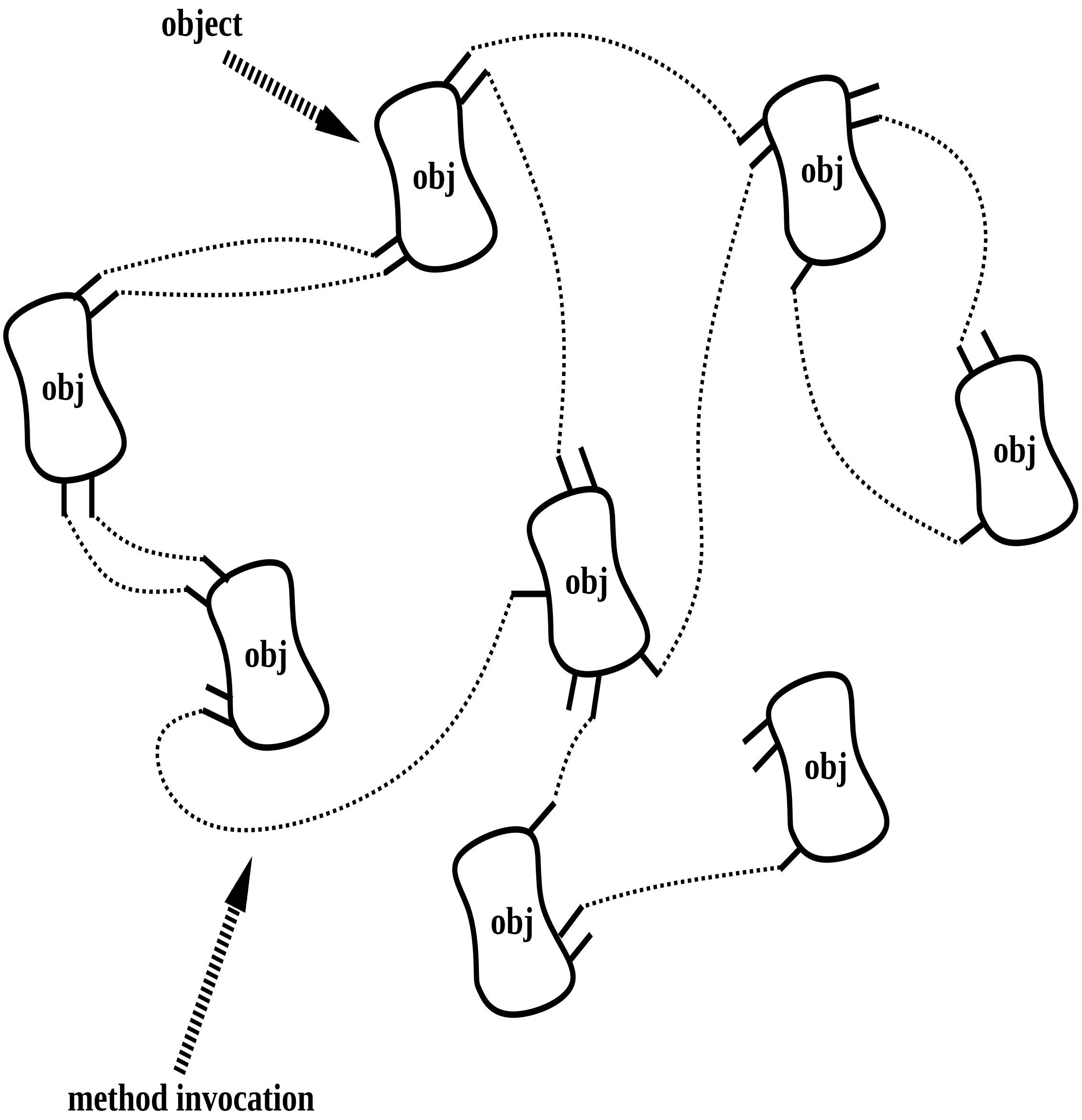
Suitable for applications in which a central issue is identifying and protecting related bodies of information (data).

Data representations and their associated operations are encapsulated in an **abstract data type**.

The components are **objects**.

The connectors are function and procedure invocations (**methods**).

Object-Oriented Style



Object-Oriented Invariants

- Objects are responsible for preserving the **integrity** (*e.g.*, some invariant) of the data representation.
- The data representation is **hidden** from other objects.

Object-Oriented Specializations

- Concurrent Objects
- Objects with Multiple Interfaces

Object-Oriented Advantages

- Because an object hides its data representation from its clients, it is possible to **change the implementation without affecting those clients**.
- Can **design systems** as collections of autonomous interacting agents.

Object-Oriented Disadvantages

- In order for one object to interact with another object (via a method invocation) the first object must know the **identity** of the second object.
  - Contrast with Pipe and Filter Style.
  - Whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it.
- Objects cause **side effect problems**. For example, if objects A and B both use object C, then B's affect on C look like unexpected side effects to A.

Implicit Invocation Style

Suitable for applications that involve loosely-coupled collection of components, each of which carries out some operation and may in the process enable other operations.

Particularly useful for applications that must be reconfigurable on the fly:

- Changing a service provider.
- Enabling or disabling capabilities.

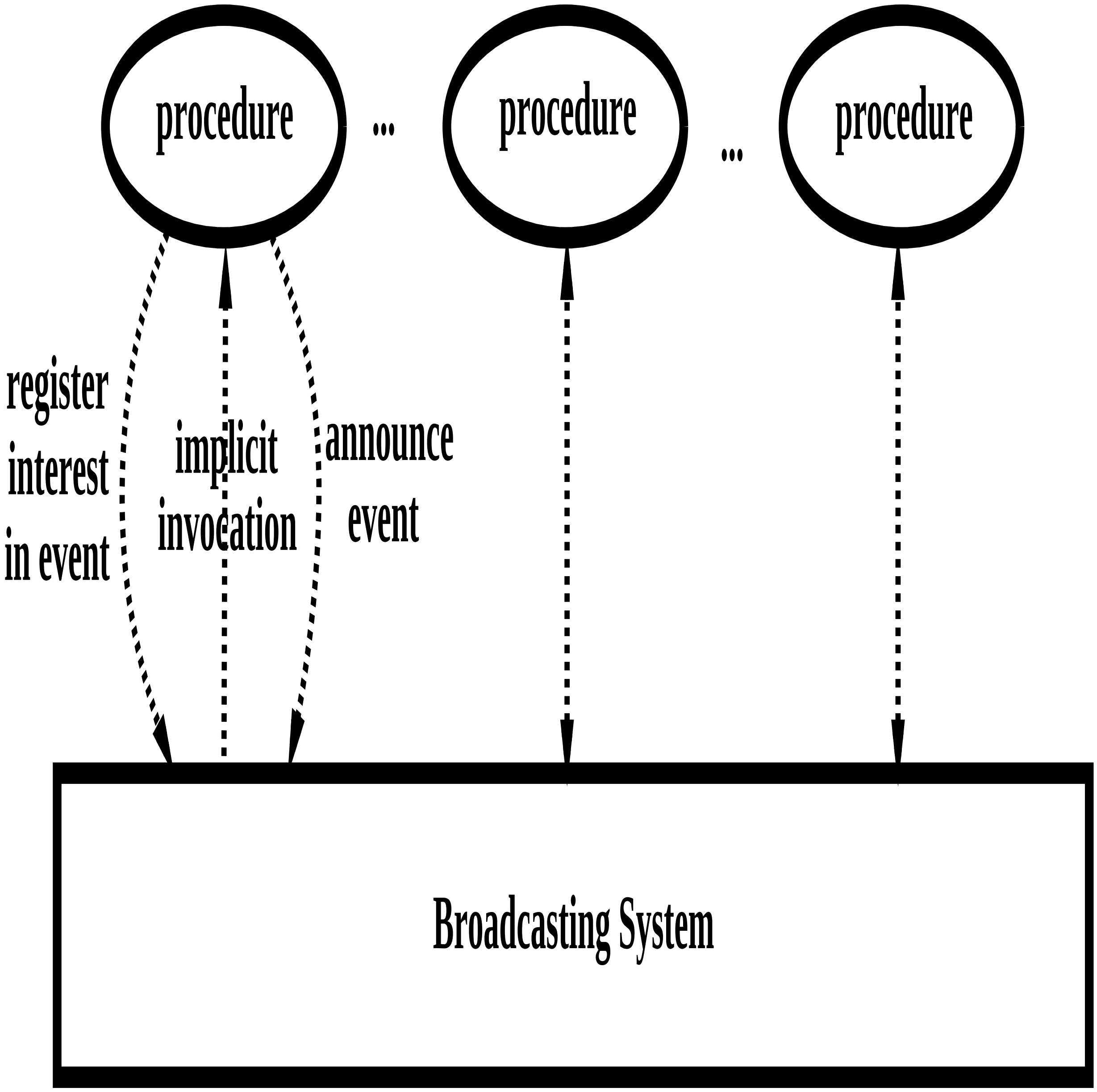
Implicit Invocation Style

Instead of invoking a procedure directly . . .

1. a **component** can announce (or broadcast) one or more events.
2. Other **components** in the system can register an interest in an event by associating a procedure with the event.
3. When an event is announced, the broadcasting system (**connector**) itself invokes all of the procedures that have been registered for the event.

Thus, an event announcement "implicitly" causes the invocation of procedures in other modules.

Implicit Invocation Style



Implicit Invocation Invariants

- Announcers of events do not know which components will be affected by those events.
- Components cannot make assumptions about the order of processing.
- Components cannot make assumptions about what processing will occur as a result of their events (perhaps no component will respond).

Implicit Invocation Specializations

- Often connectors in an implicit invocation system also include the **traditional procedure call** in addition to the bindings between event announcements and procedure calls.

Implicit Invocation Examples

- Used in programming environments to integrate tools:
  1. Debugger stops at a breakpoint and makes that announcement.
  2. Editor responds to the announcement by scrolling to the appropriate source line of the program and highlighting that line.
- Used to enforce integrity constraints in database management systems (called triggers).
- Used in user interfaces to separate the presentation of data from the applications that manage that data.

Implicit Invocation Advantages

- Provides strong support for **reuse** since any component can be introduced into a system simply by registering it for the events of that system.
- **Eases system evolution** since components may be replaced by other components without affecting the interfaces of other components in the system.

Implicit Invocation Disadvantages

- When a component announces an event:
  - it has no idea what other components will respond to it,
  - it cannot rely on the order in which the responses are invoked,
  - it cannot know when responses are finished.

Client-Server Style

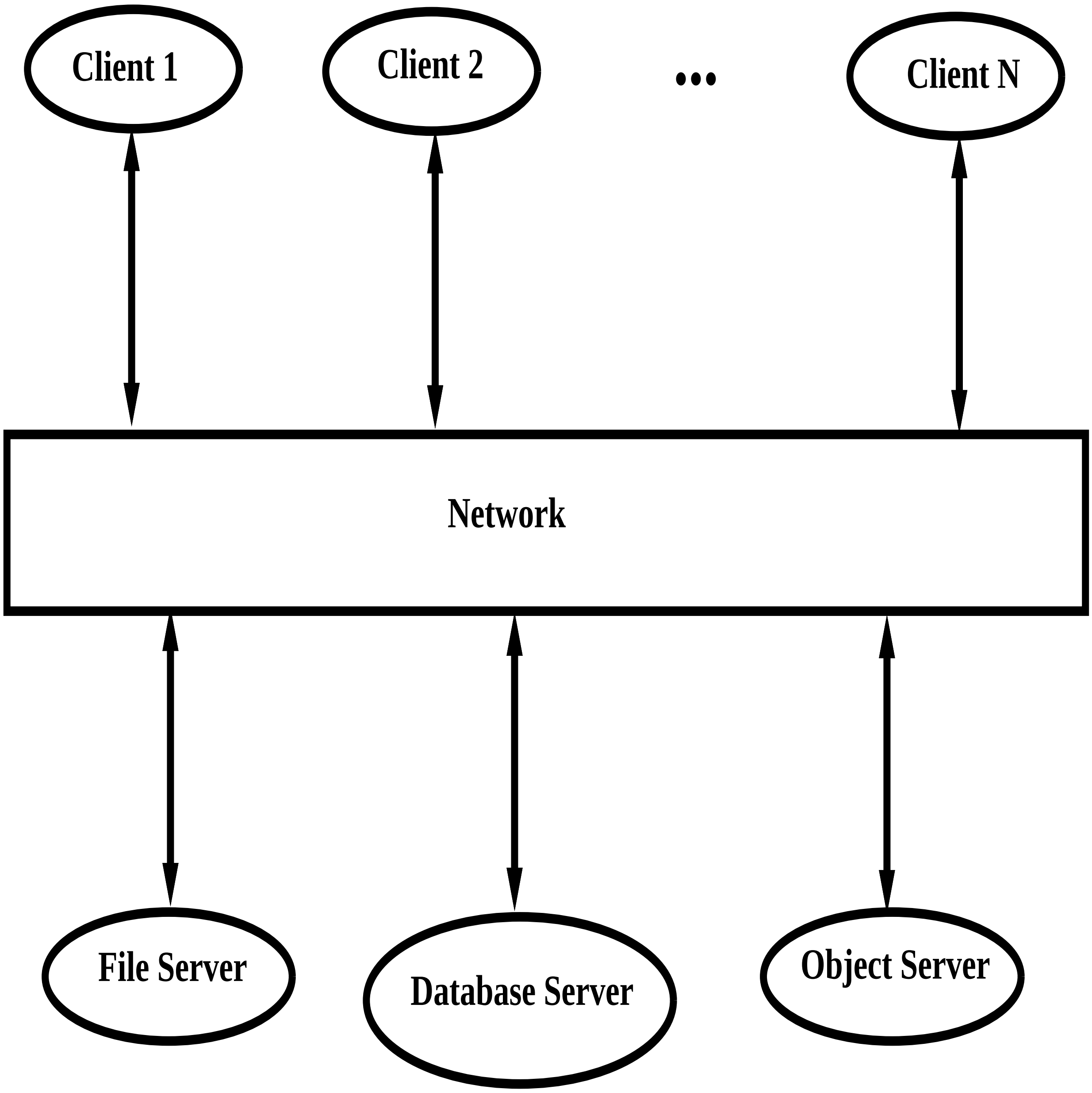
Suitable for applications that involve distributed data and processing across a range of components.

Components are either:

- *Servers:* Stand-alone components that provide specific services such as printing, data management, etc.
- *Clients:* Components that call on the services provided by servers.

The **connector** is the network, which allows clients to access remote servers.

Client-Server Style



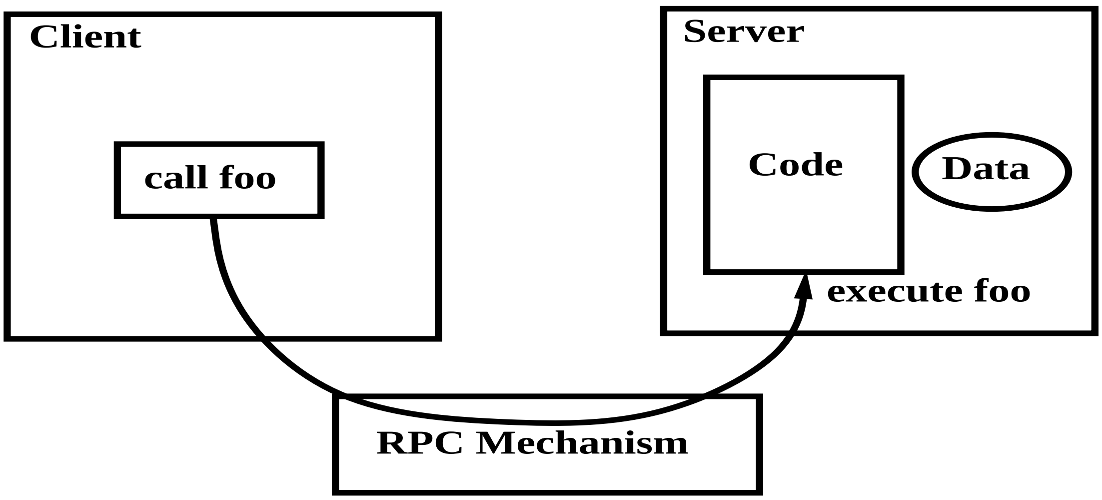
Client-Server Examples

- **File Servers:**
  - Primitive form of data service.
  - Useful for sharing files across a network.
  - The client passes request for files over the network to the file server.
- **Database Servers:**
  - More efficient use of distributing power than file servers.
  - Client passes SQL requests as messages to the DB server; results are returned over the network to the client.
  - Query processing done by the server.
  - No need for large data transfers.
  - Transaction DB servers also available.

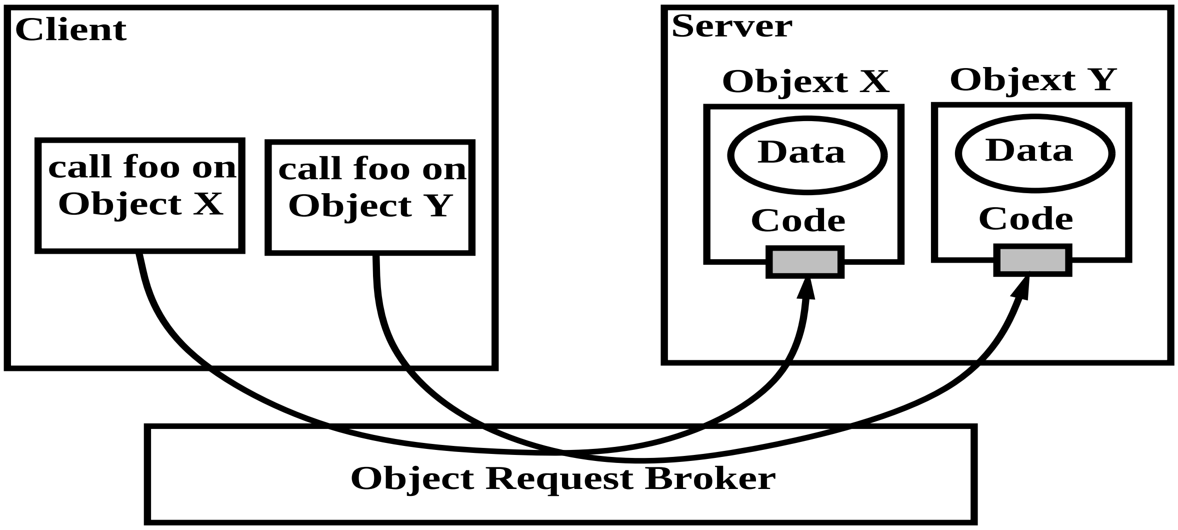
Client-Server Examples

- **Object Servers:**
  - Objects work together across machine and network boundaries.
  - ORBs allow objects to communicate with each other across the network.
  - IDLs define interfaces of objects that communicate via the ORB.
  - ORBs are the evolution of the RPC.

RPCs Versus ORBs



1) Remote Procedure Call (RPC)



2) Object Request Broker (ORB)

Client-Server Advantages

- Distribution of data is straightforward,
- transparency of location,
- mix and match heterogeneous platforms,
- easy to add new servers or upgrade existing servers.

Client-Server Disadvantages

- No central register of names and services – it may be hard to find out what services are available

Layered Style

Suitable for applications that involve distinct classes of services that can be **organized hierarchically**.

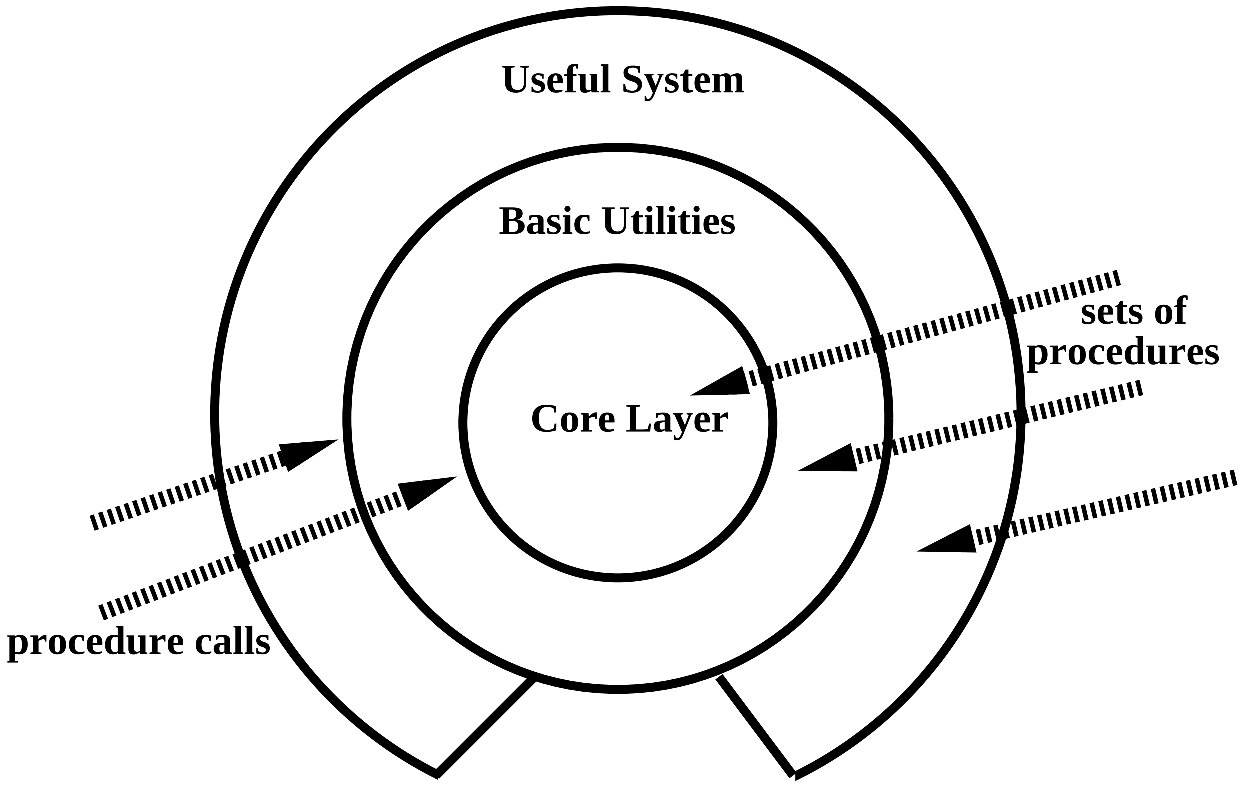
Each layer provides service to the layer above it and serves as a client to the layer below it.

Only carefully selected procedures from the inner layers are made available (exported) to their adjacent outer layer.

Components are typically collections of procedures.

Connectors are typically procedure calls under restricted visibility.

Layered Style

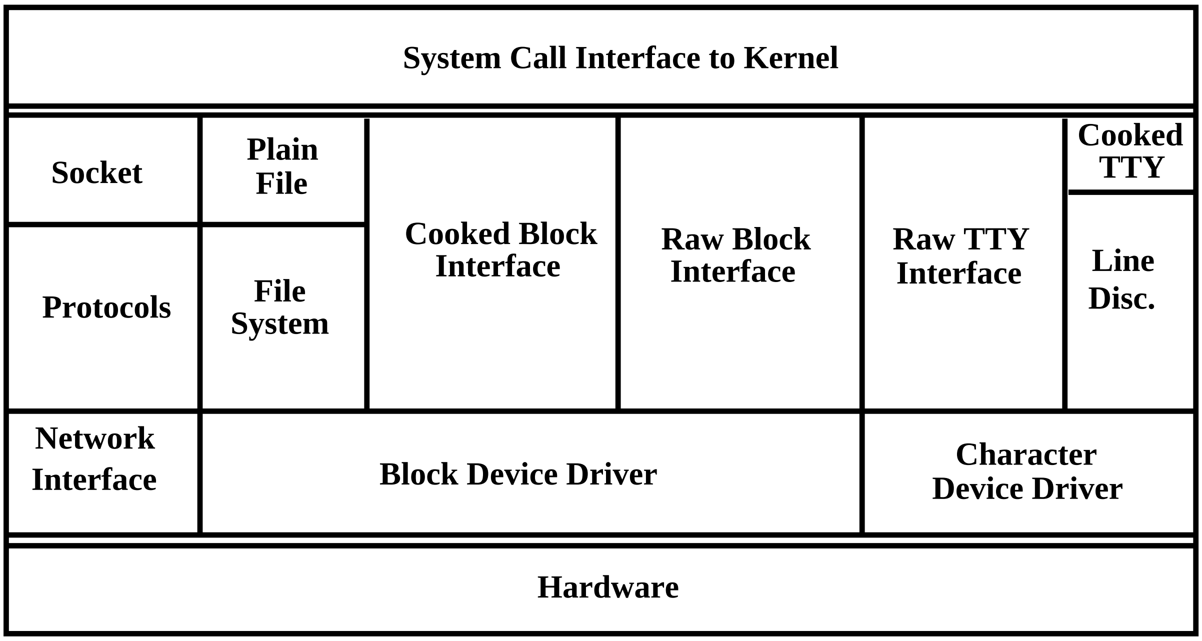


Layered Style Specializations

- Often exceptions are be made to **permit non-adjacent layers to communicate directly**. This is usually done for efficiency reasons.

Layered Style Examples

- **Layered Communication Protocols**
  - Each layer provides a substrate for communication at some level of abstraction.
  - Lower levels define lower levels of interaction, the lowest level being hardware connections (physical layer).
- **Operating Systems** (*e.g.*, Unix below [Silberschatz and Peterson, 1988])



Layered Style Advantages

- Support **design** based on increasing levels of abstraction.
- They support **enhancement** since changes to the function of one layer affects at most two other layers.
- They support **reuse**, since different implementations (with identical interfaces) of the same layer can be used interchangeably.

Layered Style Disadvantages

- Not all systems are easily structured in a layered fashion.
- Performance requirements may force the coupling of high-level functions to their lower-level implementations.

Repository Style

Suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information.

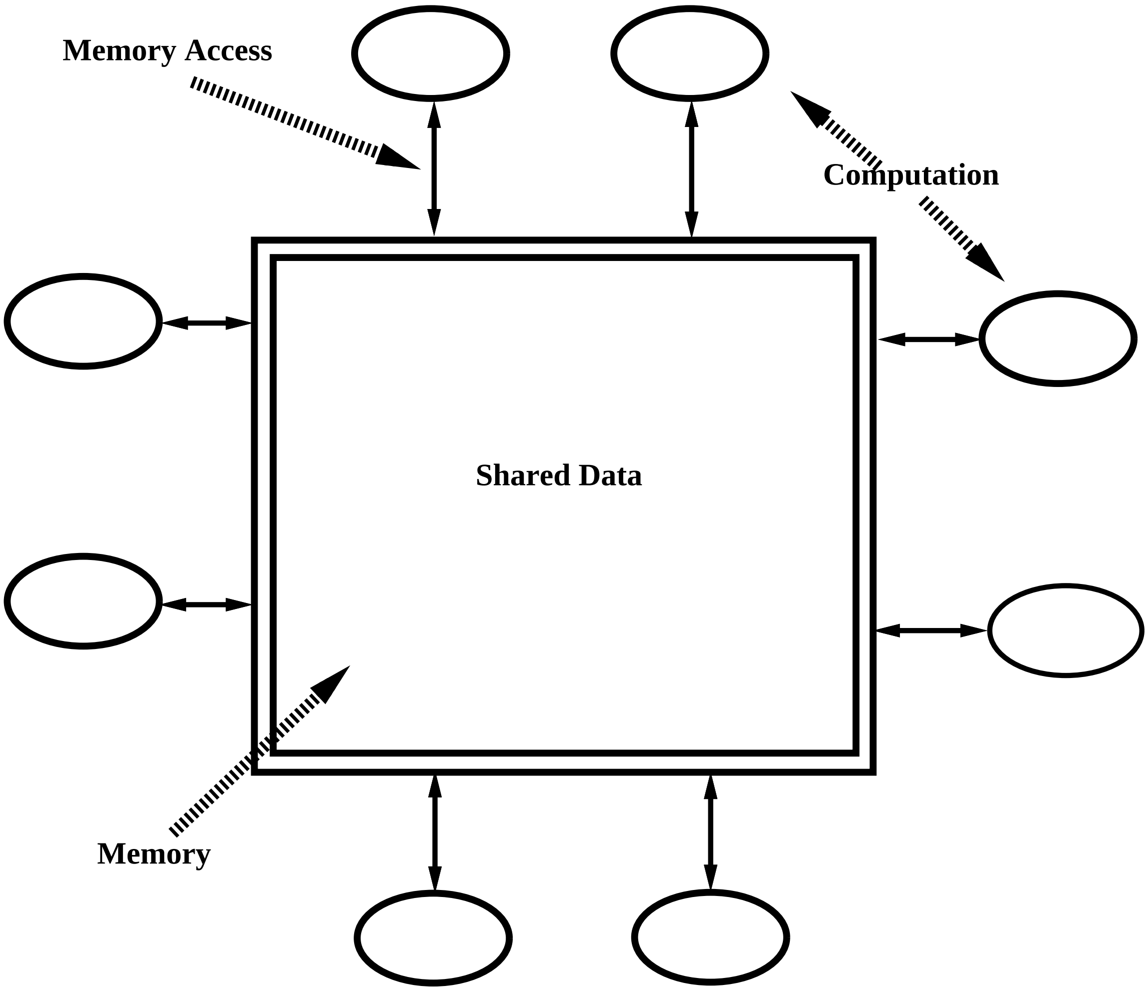
Typically the information must be manipulated in a variety of ways. Often long-term persistence is required.

Components:

- A central data structure representing the correct state of the system.
- A collection of independent components that operate on the central data structure.

Connectors are typically procedure calls or direct memory accesses.

Repository Style



Repository Style Specializations

- Changes to the data structure trigger computations.
- Data structure in memory (persistent option).
- Data structure on disk.
- Concurrent computations and data accesses.

Repository Style Examples

- Information Systems
- Programming Environments
- Graphical Editors
- AI Knowledge Bases
- Reverse Engineering Systems

Repository Style Advantages

- **Efficient** way to store large amounts of data.
- **Sharing** model is published as the repository schema.
- Centralized **management**:
  - backup
  - security
  - concurrency control

Repository Style Disadvantages

- Must **agree on a data model** a priori.
- Difficult to **distribute data**.
- Data **evolution is expensive**.

Interpreter Style

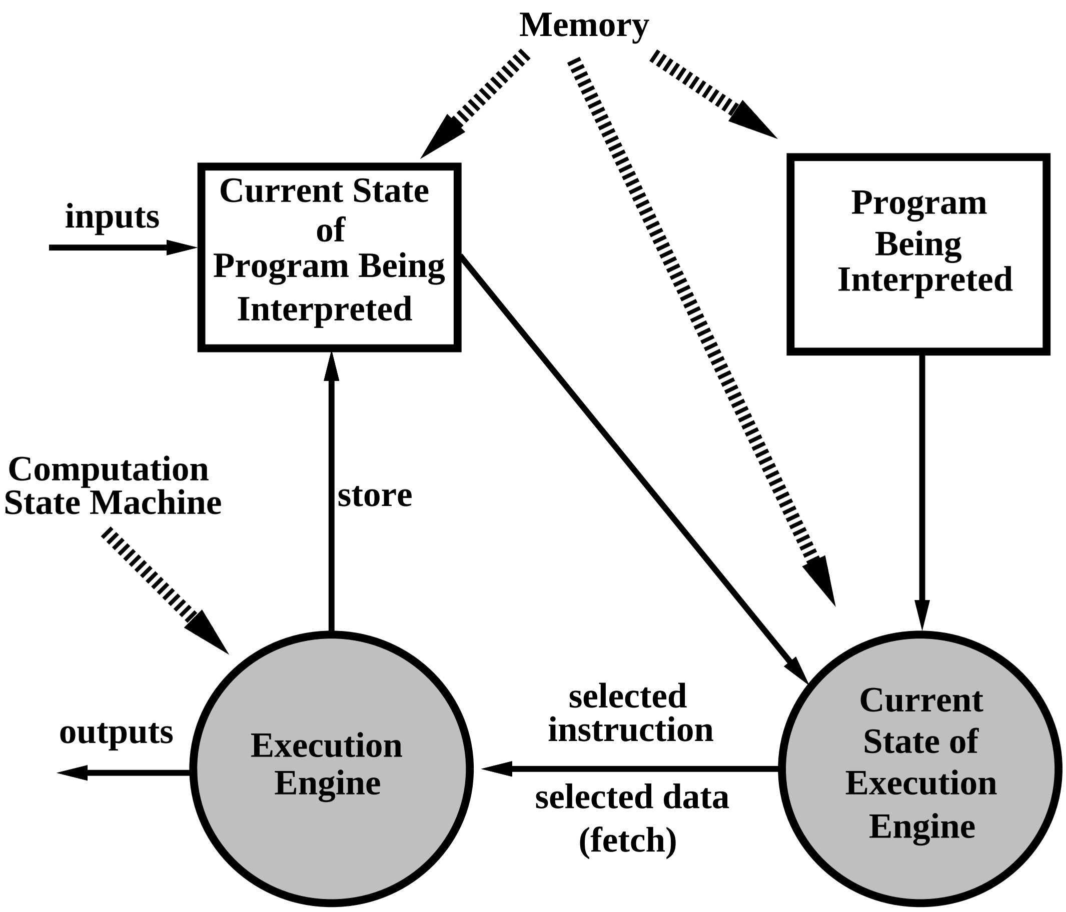
Suitable for applications in which the most appropriate language or machine for executing the solution is not directly available.

Components include one state machine for the execution engine and three memories:

1. current state of the execution engine
2. program being interpreted
3. current state of the program being interpreted

Connectors are typically procedure calls or direct memory accesses.

Interpreter Style

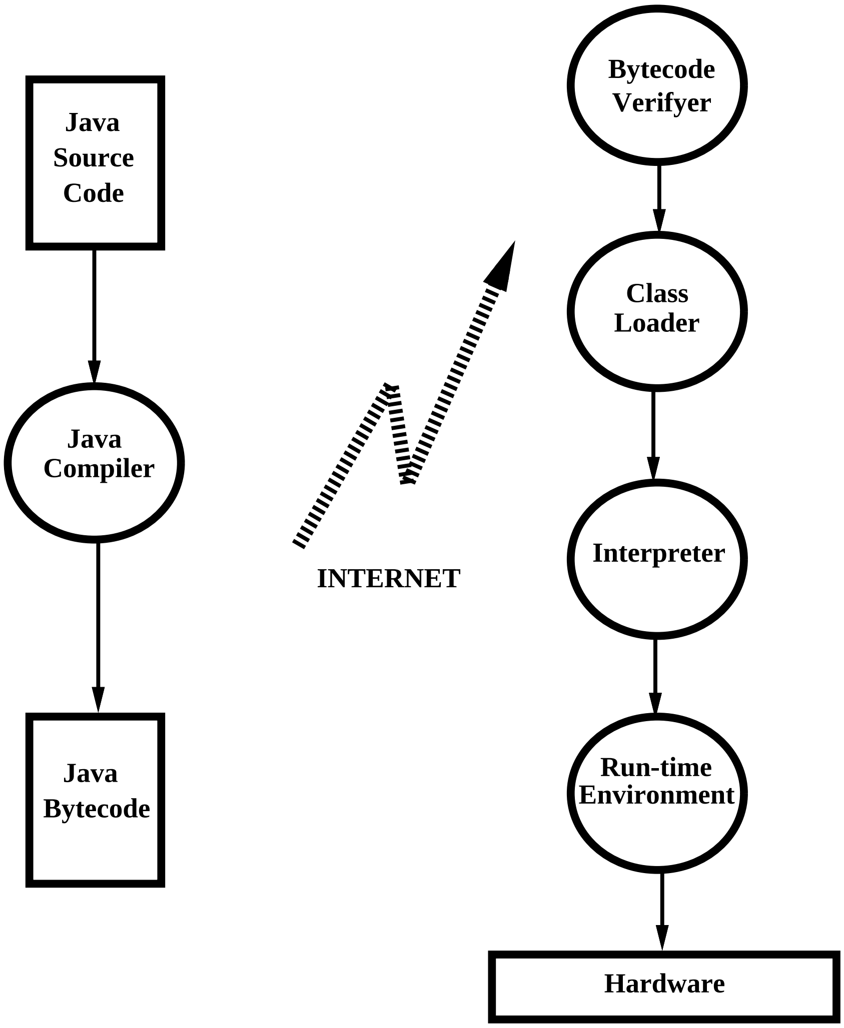


Interpreter Style Examples

- Programming Language Compilers (*e.g.*, Java, OOT)
- Rule Based Systems (*e.g.*, Prolog, Telos)
- Scripting Languages

Interpreter Style Advantages

- **Simulation** of non-implemented hardware.
- Facilitates **portability** of application or languages across a variety of platforms (*e.g.*, Java).



Interpreter Style Disadvantages

- Extra level of indirection **slows** down execution. Both Java and OOT have options to compile code.

Process-Control Style

Suitable for applications whose purpose is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values.

Components:

- **Process Definition** includes mechanisms for manipulating some process variables.
- **Control Algorithm** for deciding how to manipulate process variables.

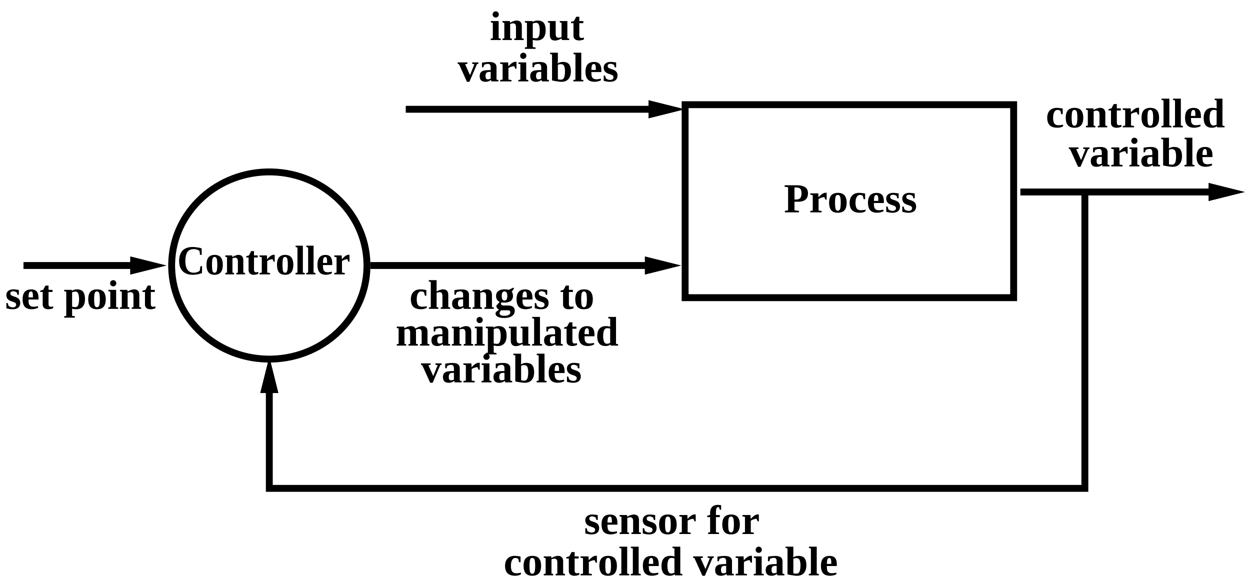
Process-Control Style

Connectors are the data flow relations for:

- **Process Variables:**
  - **Controlled variable** whose value the system is intended to control.
  - **Input variable** that measures an input to the process.
  - **Manipulated variable** whose value can be changed by the controller.
- **Set Point** is the desired value for a controlled variable.
- **Sensors** to obtain values of process variables pertinent to control.

Process-Control Style

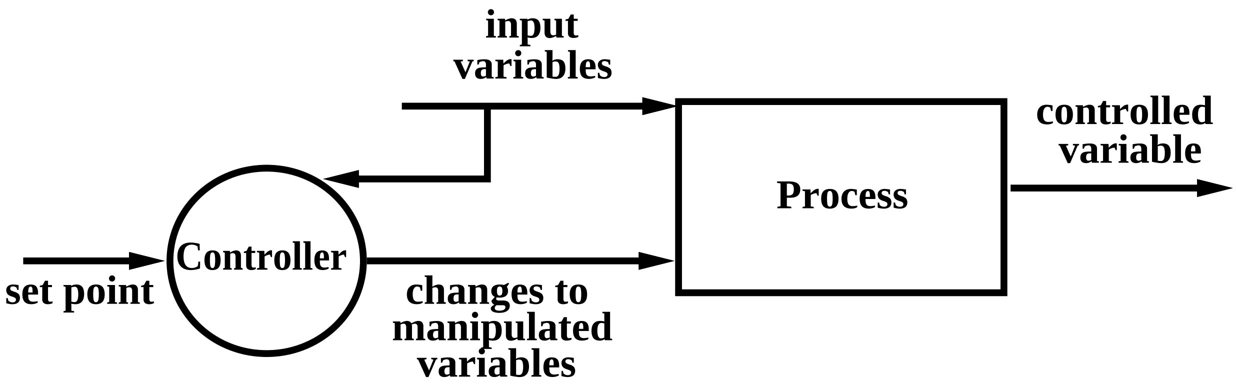
Feed-Back Control System: The controlled variable is measured and the result is used to manipulate one or more of the process variables.



Process Control Specializations

- **Open-Loop System:** Information about process variables is not used to adjust the system.

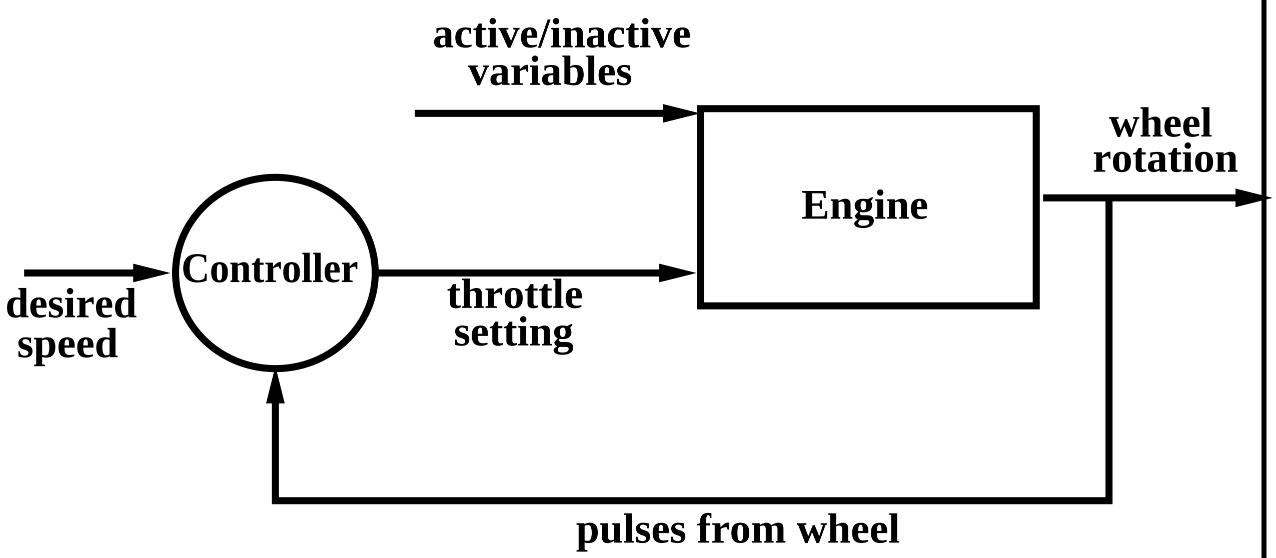
- **Feed-Forward Control System:** The controlled variable is measured and anticipated disturbances are compensated for without waiting for changes in the controlled variable to be visible.



Process Control Examples

Real-Time System Software for Controlling:

- Automobile Anti-Lock Brakes
- Nuclear Power Plants
- Automobile Cruise-Control

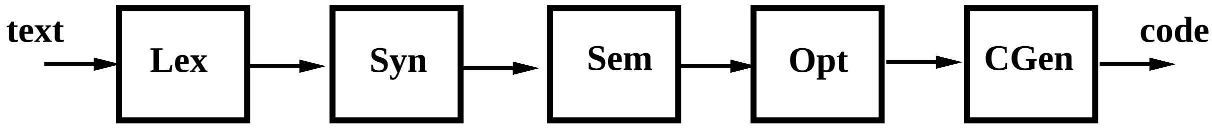


Case Study: Compiler Architecture

Case Study: Compiler Architecture

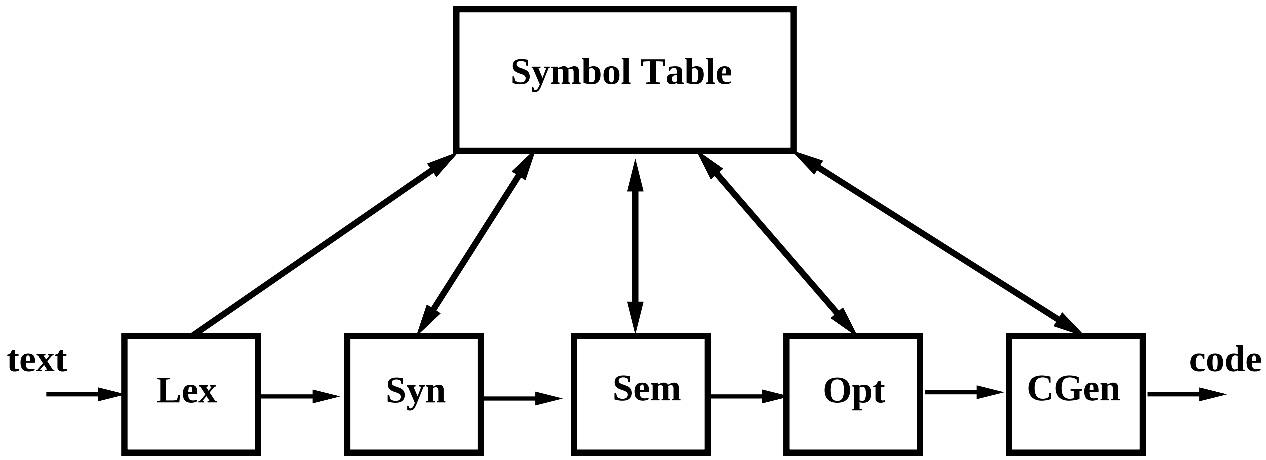
The architecture of a system can change in response to improvements in technology. This can be seen in the way we think about compilers.

In the 1970s, compilation was regarded as a sequential (batch sequential or pipeline) process:



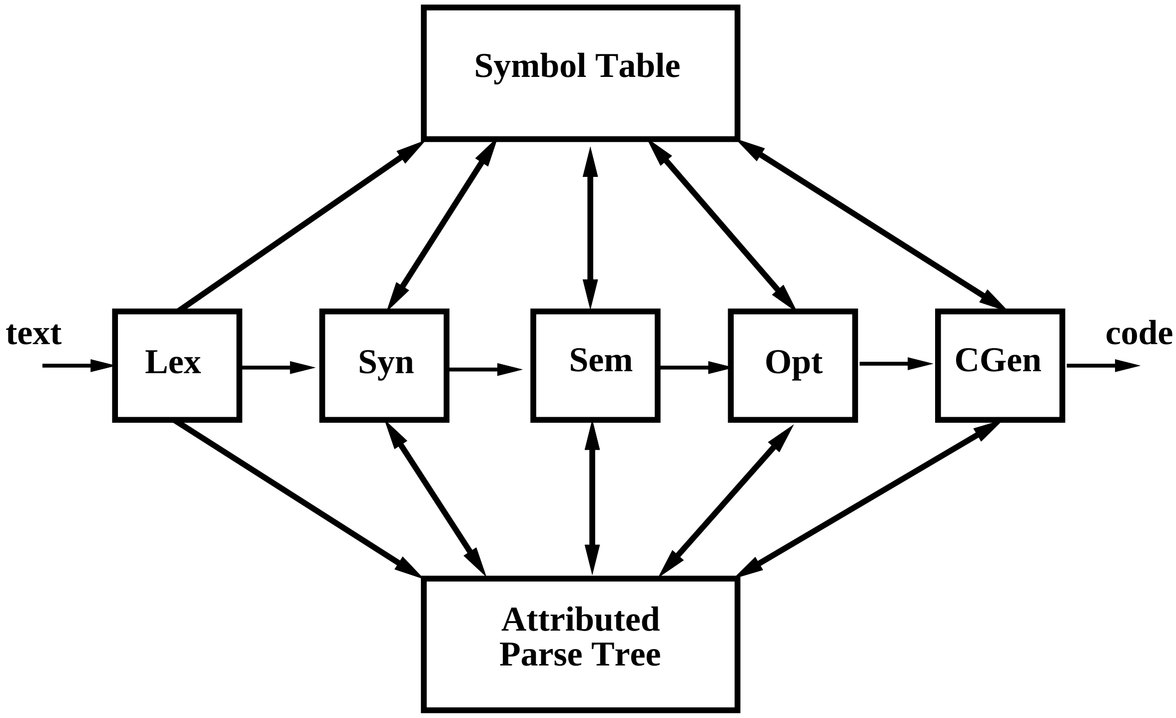
Case Study: Compiler Architecture

Most compilers create a separate symbol table during lexical analysis and used or updated it during subsequent passes.



Case Study: Compiler Architecture

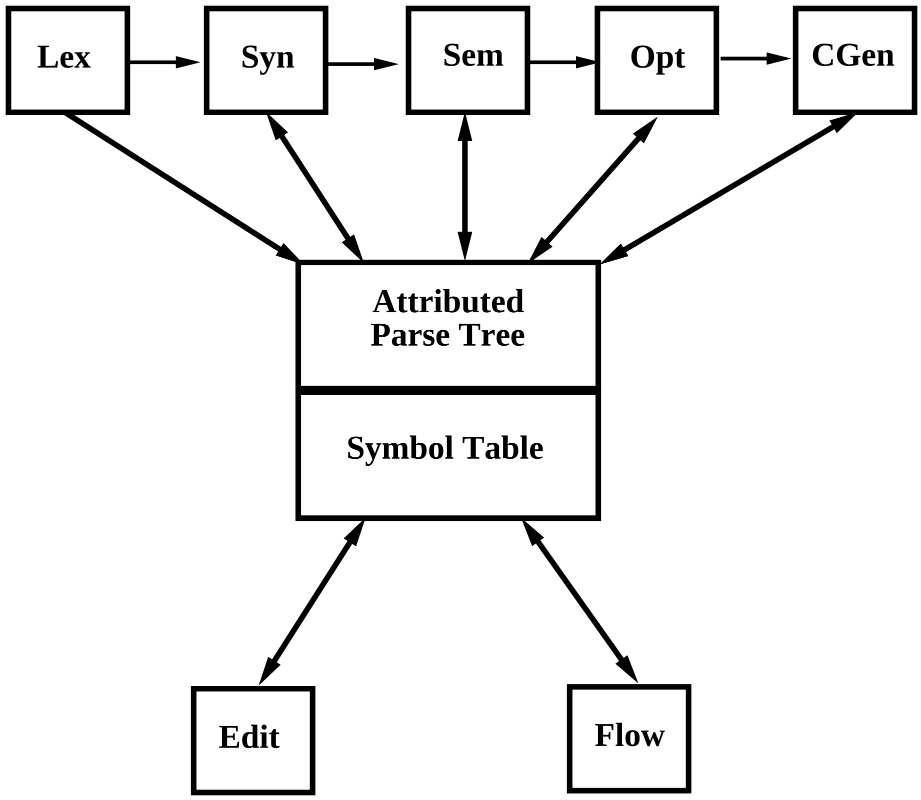
Later, in the mid 1980s, increasing attention turned to the intermediate representation of the program during compilation (*e.g.*, attributed parse tree).



Case Study: Compiler Architecture

The new view accommodates various tools (*e.g.*, syntax-directed editors) that operate on the internal representation rather than the textual form of a program.

Architectural shift to a **repository** style, with elements of the **pipeline** style, since the order of execution of the processes is still predetermined.



The Future of Software Architecture

The Future of Software Architecture

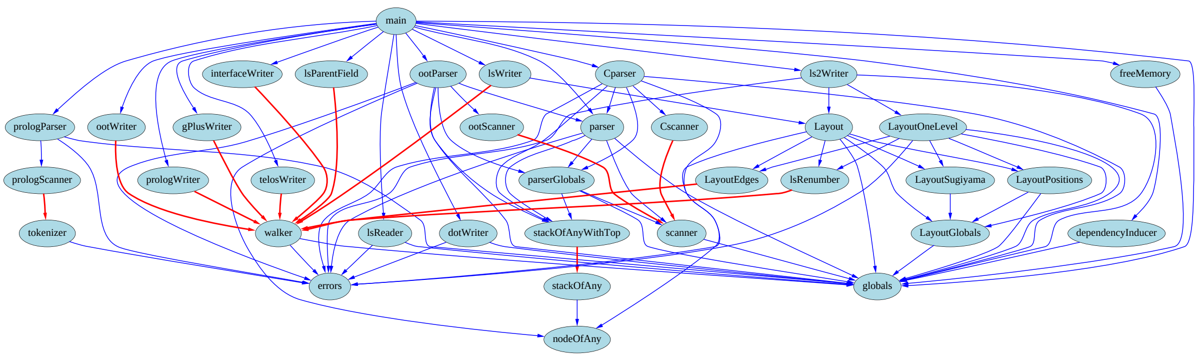
The understanding of the emerging **vocabulary** of software architecture is a significant intellectual tool for the software engineer.

We expect to see significant advances in a number of related areas including:

- **Taxonomies** of architectural styles.
- **Formal** models for characterizing and analyzing architectures.
- **Notations** for describing architectural designs.
- **Tools** and environments for developing architectural designs.
- Techniques for **extracting** architectural information from existing code.
- Better understanding of the role of architectures in the **life-cycle process**.

Software Structure Topics

- Terminology and Motivation
- Theory of Modularity and Information Hiding
- Software Modularization
- ASF Formalism
- Programming Language Support for Modularity and Information Hiding
- Module Interconnection Languages
- Styles of Software Structure



Terminology and Motivation

Software Structure

Fact: Large software systems have hundreds of components (modules, classes) and dependencies (import, export, inherit) between them.

Problem: Important aspects of the structure are often buried inside the numerous components and dependencies.

Assumption: Understanding the structure of a software system is crucial for its successful implementation, testing, and maintenance.

State of Practice

Developers create a *mental model* of what they believe are the important aspects of the software structure.

These models are often represented as informal diagrams.

Problem: Mental models may be:

- incorrect,
- out of date,
- lacking a concrete representation,
- difficult for new developers to create.

State of Research

Research in *software structure* aims at helping developers understand and manage the structure of large software systems at varying levels of detail.

Current research trends include:

- notations for specifying software structure (*e.g.*, MILs, ADLs),
- techniques for visualizing software structure (*e.g.*, layout, elision),
- reverse engineering techniques for extracting structural information via source code analysis and design metrics.

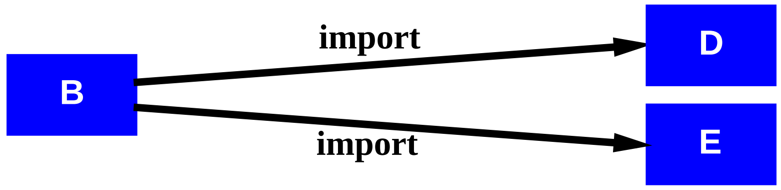
Software Structure

Software structure is used to specify the:

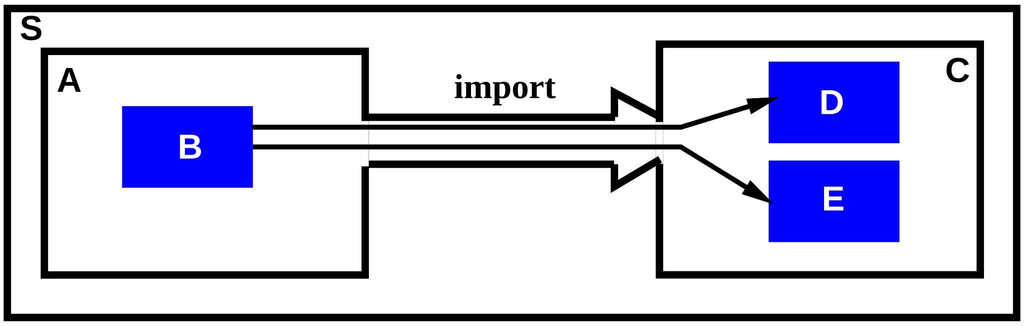
- decomposition of a system into a subsystem hierarchy,
- interfaces of components (subsystems, modules, classes),
- dependencies between components.

Limitation: Software structure say little about the run-time computational elements of a system and their interactions.

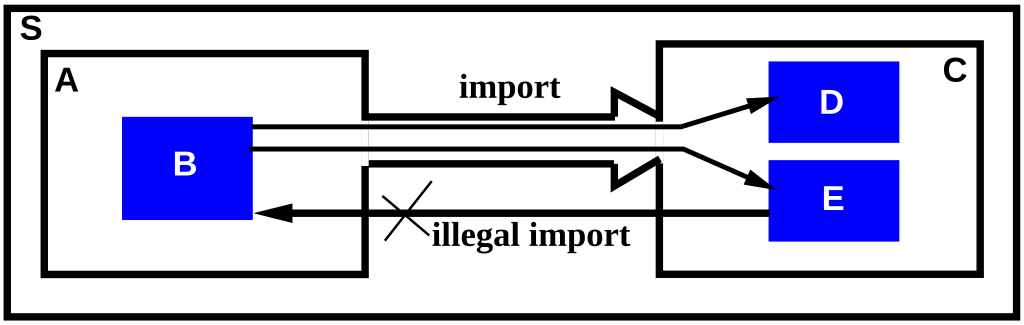
Examples of Software Structures



Module Interconnections

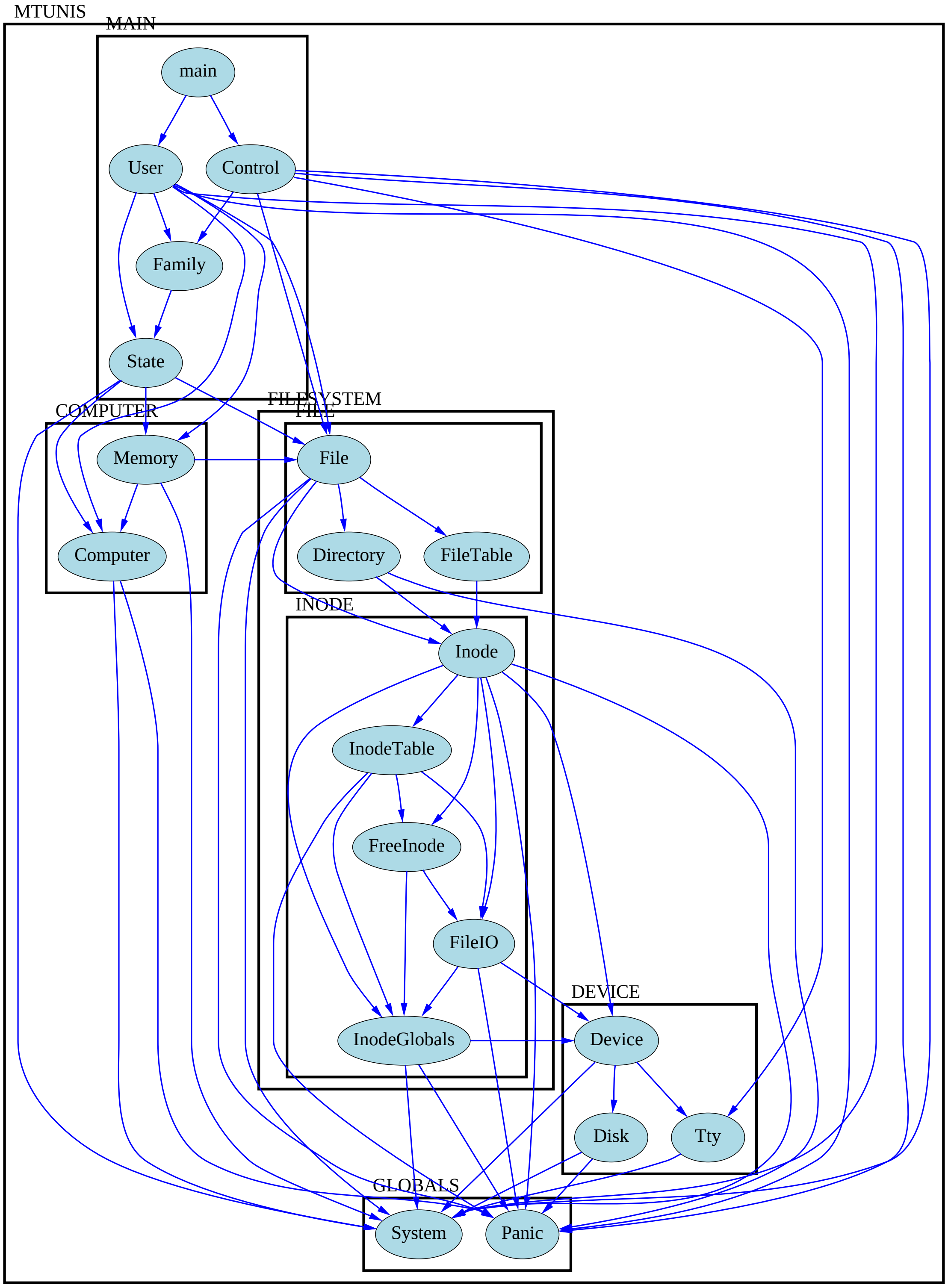


Adding Composite Modules (subsystems) and Composite Access Permissions

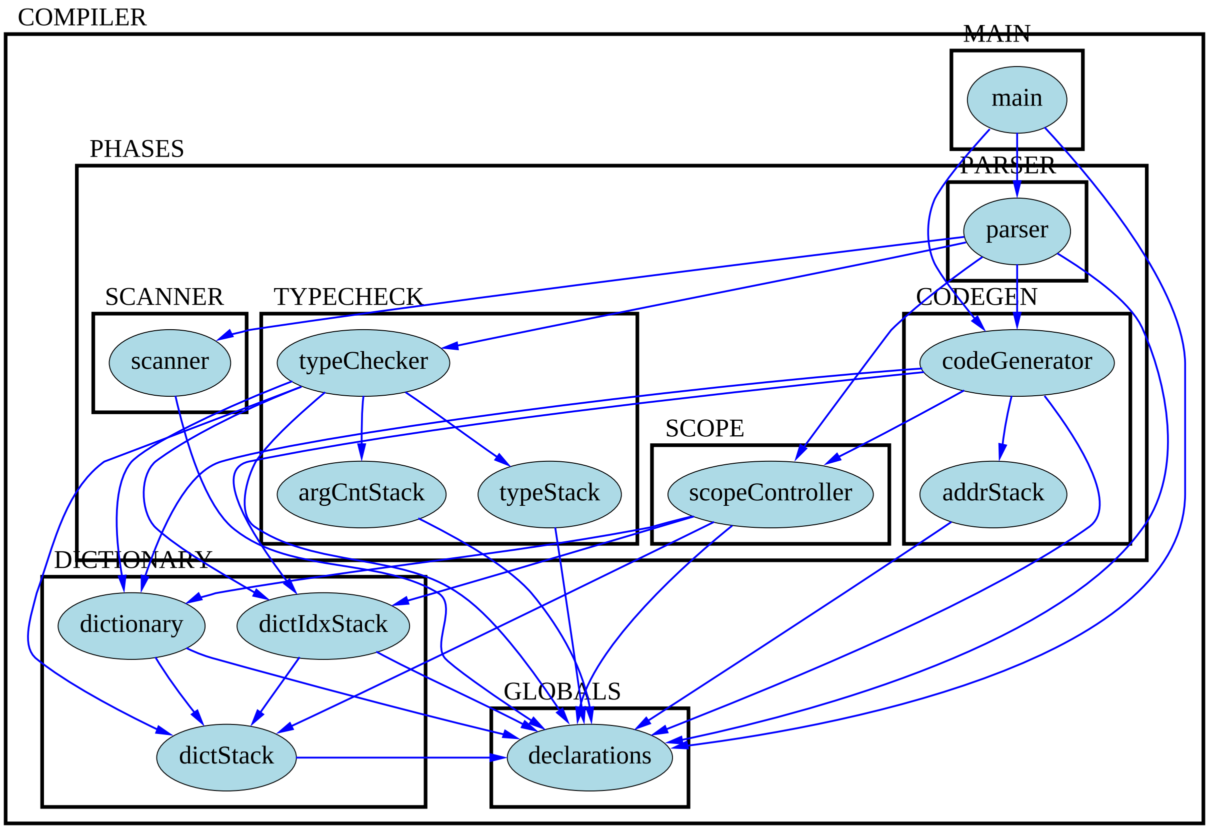


Ill-formed Software Structure

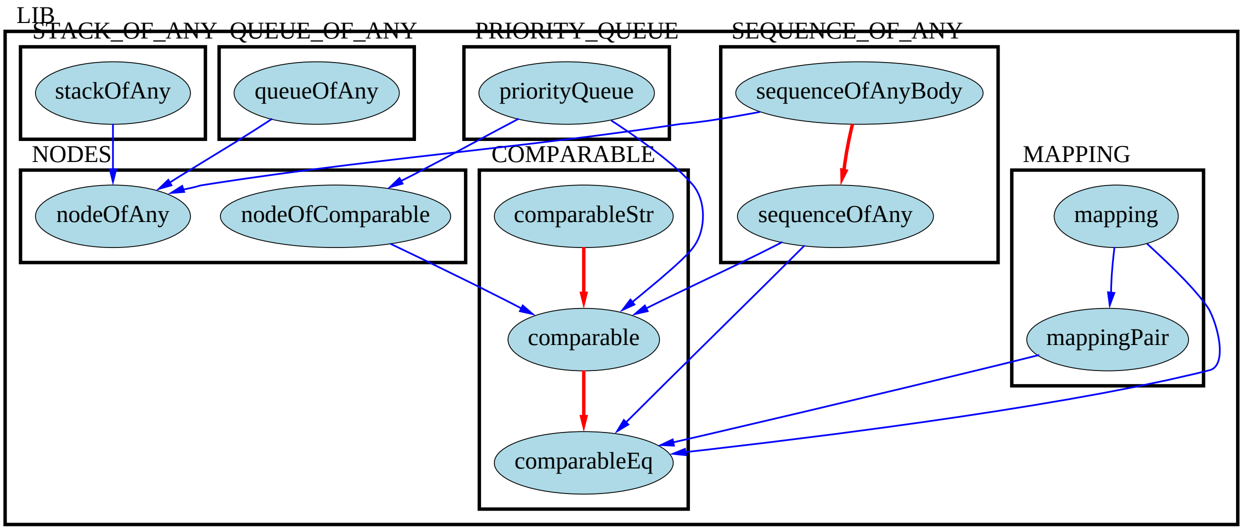
Structure of an OS



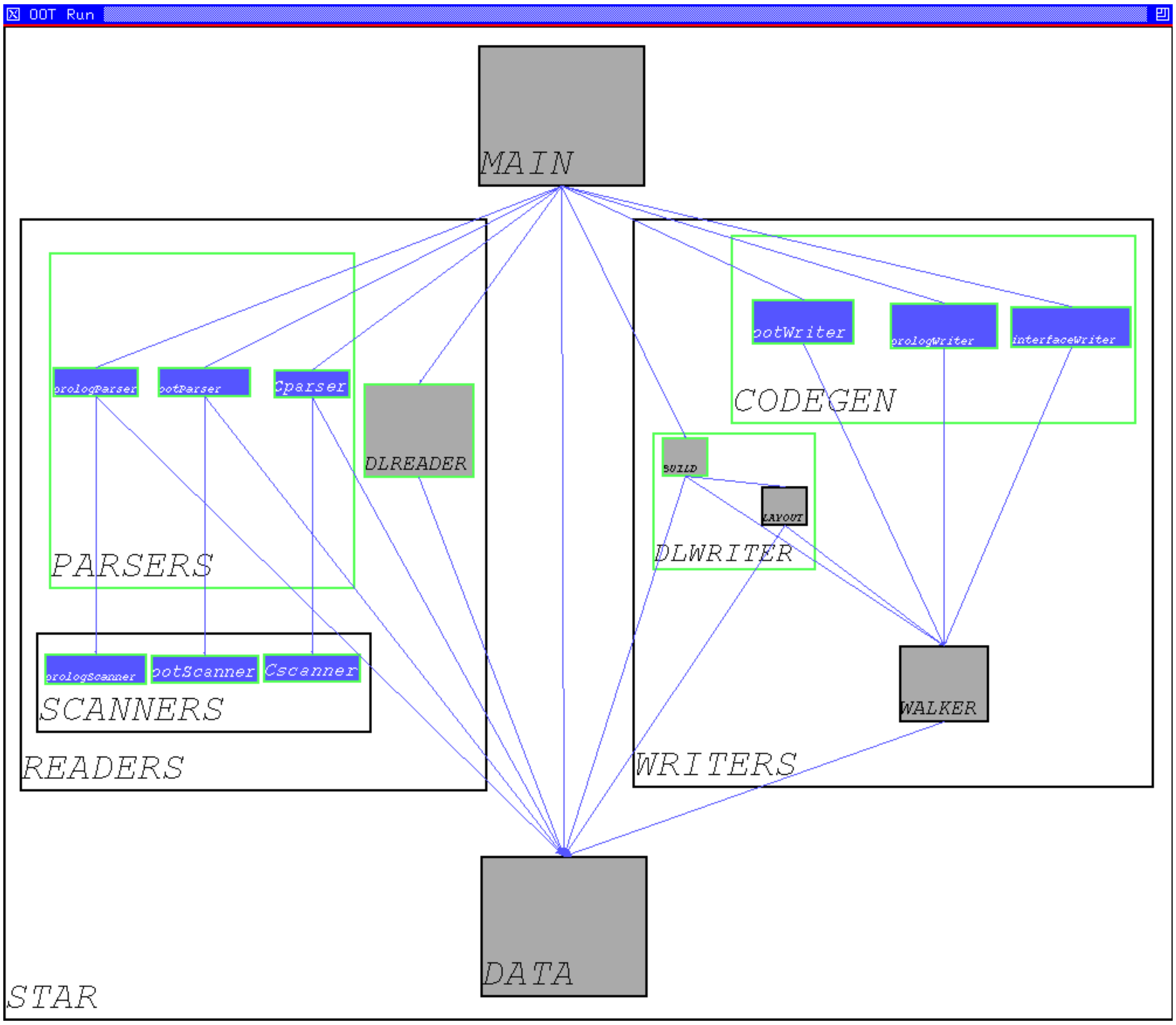
Structure of a Compiler



Structure of a Data Structures Library



Structure of a Reverse Engineering System



Theory of Modularity and Information Hiding

Theory of Modularity and Information Hiding

The basic idea of modular design is to decompose a system into modules, each of which defines some collection of programming language entities (*e.g.*, variable, procedure) called resources.

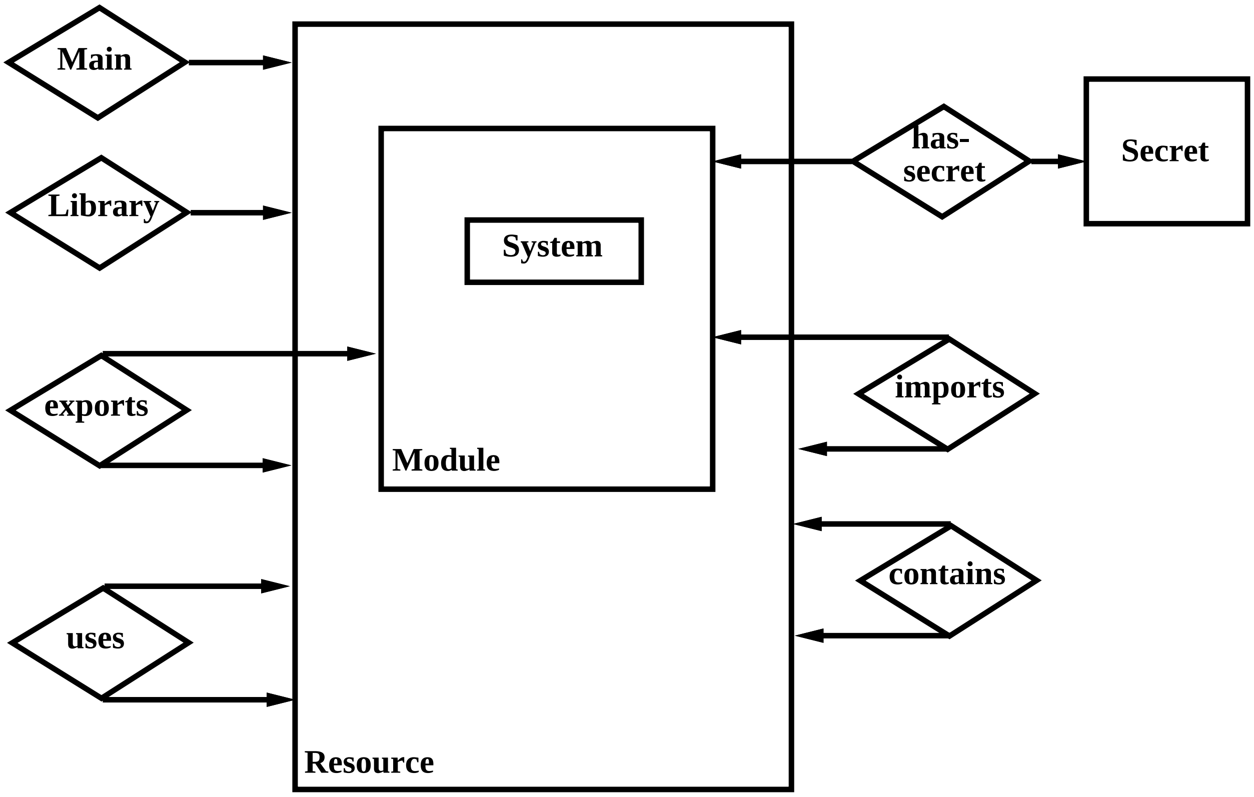
A **modular design** is a tuple:

$$(Modules, Resources, Decomposition, Dependencies)$$

where *Modules* is the set of modules and *Resources* is the set of resources in a system.

Based on material in [Parnas, 1972, Lamb and Schneider, 1992]

E/R Model of Theory



Module Decomposition

To be easy to comprehend, a module should be:

- **small**: no more than a few hundred lines of code;
- **cohesive**: the components of the module should be closely related;
- **loosely coupled**: the module should have a few narrow interactions with other modules.

Decomposition is a tuple:

$$(\mathit{System}, \mathit{Secret}, \mathit{has\text{-}secret}, \mathit{contains})$$

- $\mathit{System} \in \mathit{Module}$
- *contains* is a relation that defines a tree rooted at *System* (subsystem hierarchy)

$$\mathit{contains} \subseteq \mathit{Resource} \times \mathit{Resource}$$

Module Secrets

A module hides one or more design decisions, called **secrets**. Examples of secrets:

- data structure representations,
- algorithms,
- policies,
- one or more of the modules its contains.

Secret is the set of all secrets in a system.

has-secret is a relation on *Module* to *Secret*:

$$\mathit{has\text{-}secret} \subseteq \mathit{Module} \times \mathit{Secret}$$

Each secret is the secret of at least one module:

$$\mathit{Domain}(\mathit{has\text{-}secret}^{-1}) = \mathit{Secret}$$

Module Dependencies

Designers should restrict the module dependencies structure to ensure the simplicity of the design and ease of decomposition.

Dependencies is a tuple:

$$(imports, exports, Main, uses, Library)$$

- $exports \subseteq (contains \mid Module \times Resource)$ (resources a module makes available)
- $import \subseteq Module \times Resource$ (exported resources a module may use)
- $Main \in Resource$
- $uses \subseteq Resource \times Resource$ (dependencies between resources)
- $Library \subset Resource$ (reusable resources)

Module Dependencies

All resources, except possibly those exported by library modules, are reachable via the *uses* relation from *Main*:

$\forall r \in \mathit{Resources} \cdot (r \in \mathit{Library} \vee \mathit{Main}\ \mathit{uses}^*\ r)$

A resource *R1* can **see** another resource *R2* if *R1* is a module and *R1* imports *R2* or *R1* is not a module and the module that contains it imports *R2*.

A resource may only use another resource if it can see it:

$$\mathit{use} \subseteq \mathit{see}$$

Software Modularization

Software Modularization

The goal of the modularization process is to partition the set of modules of a system in such as way as to result in subsystems that exhibit a:

- **high** degree of internal **cohesion**,
- **low** degree of external **coupling** with other subsystems

Material on Software Modularization based on [Müller et al., 1992]

Software Modularization Process

1. Determine the relations between atomic components (modules),
2. compute the interconnection strengths between modules,
3. set parameters and compute design metrics,
4. based on the metrics, produce a variety of views of the module structure,
5. based on the intuition gained from producing the views, create the subsystem decomposition hierarchy.

Module Interconnection Strength

IS(m_i,m_j) is the **interconnection strength** between an ordered pair of distinct modules m_i and m_j $(i \neq j)$.

IS(m_i,m_j) depends on the number of:

- function calls,
- variable references,
- type references,
- macro references,
- ...

that exist between modules m_i and m_j.

Module Interconnection Strength

For example, assume that:

- 2 function calls,
- 5 variable references,
- 2 type references,
- 0 macro references,

exist between modules m_i and m_j. Then the interconnection strength is:

$$\mathrm{IS}(m_i, m_j) = 2 + 5 + 2 + 0 = 9$$

Note, in general: $\mathbf{IS}(m_i, m_j) \neq \mathbf{IS}(m_j, m_i)$

Module Interconnection Strength

What about assigning weights to the various dependencies?:

$$\text{IS}(m_i,m_j) = \text{w}_{fc}\times|\text{fc}(m_i,m_j)| + \text{w}_{vr}\times|\text{vr}(m_i,m_j)| + \text{w}_{tr}\times|\text{tr}(m_i,m_j)| + \text{w}_{mr}\times|\text{mr}(m_i,m_j)|$$

fc(m_i,m_j) is the **set** of ordered pairs of function calls from entities in m_i to functions in m_j.

$|$**fc**$(m_i,m_j)|$ is the **cardinality** of set fc(m_i,m_j).

w_{fc}: **weight** assigned to **function** calls.
w_{vr}: **weight** assigned to **variable** references.
w_{tr}: **weight** assigned to **type** references.
w_{mr}: **weight** assigned to **macro** references.
(configurable parameters)

Strongly Coupled Modules

Module m_i is strongly coupled with module m_j when:

$$\textbf{IS}(m_i, m_j) \geq T_{sc}$$

where T_{sc} is the configurable strong coupling **threshold** value.

Clustering Guideline: *Strongly coupled modules and more likely to be assigned to the same subsystem.*

Weakly Coupled Modules

Module m_i is weakly coupled with module m_j when:

$$\mathbf{IS}(m_i,m_j) \leq T_{wc}$$

where T_{wc} is the configurable weak coupling **threshold** value.

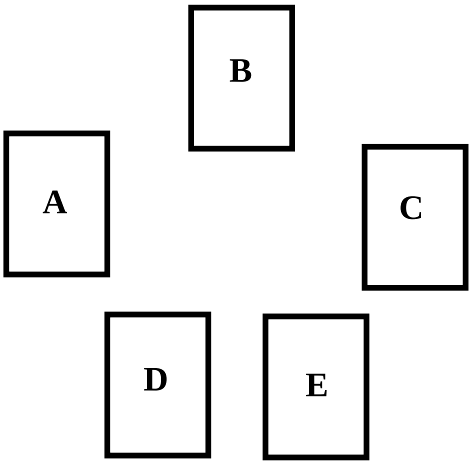
Clustering Guideline: *Weakly coupled modules and more likely to be assigned to separate subsystems.*

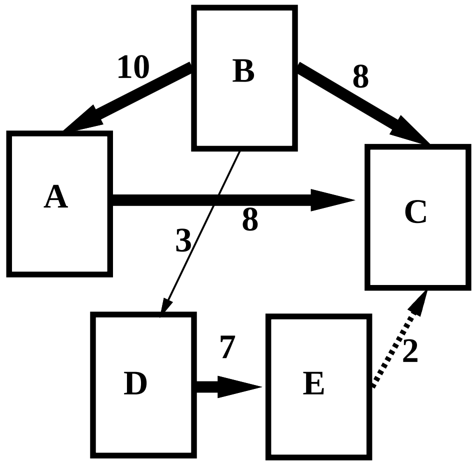
All other interconnection strengths are neither strong nor weak:

$$T_{wc} < \mathrm{IS}(m_i,m_j) < T_{sc}$$

Module Coupling Example

$$T_{sc} = 7 \text{ and } T_{wc} = 2$$





Bold Edges: Strong interconnections

Dashed Edges: Weak interconnections

Module External Strength

The **external strength** ES(m) of a module m is defined as the sum of the interconnection strengths of that module to the rest of the modules in the system.

$$ES(m) = \Sigma_{i=1}^{k} IS(m, m_i)$$

where:

- k is the number of modules in the system,
- M is the set of modules in the system

$$M = \bigcup_{i=1}^{k} \{m_i\}$$

- $m \in M$

Central and Fringe Modules

A module m is **central** when it is heavily depended on by other modules:

$$\mathbf{ES}(m_i) \geq T_c$$

where T_c is the configurable central module **threshold** value.

A module m is **fringe** when it has a peripheral role in the system:

$$\mathbf{ES}(m_i) \leq T_f$$

where T_f is the configurable fringe module **threshold** value.

Clustering Guideline: *Central and fringe modules are normally assigned to separate subsystems.*

Omnipresent Modules

A module m is **omnipresent** when it has many other modules of the system depending on it.

$$T_{op} \leq |\mathbf{clients}(m)|$$

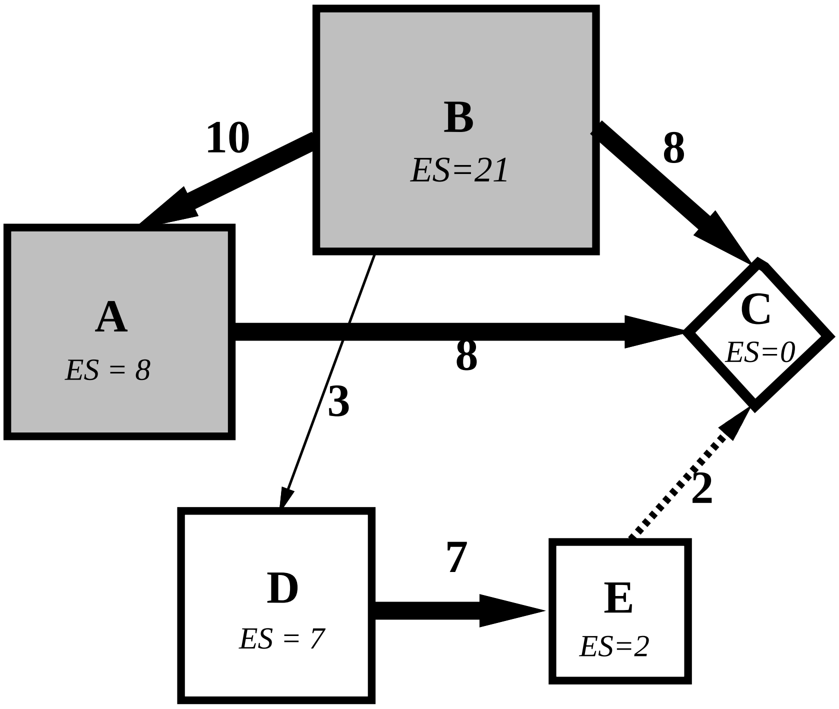
- T_{op} is the configurable omnipresent module **threshold** value,
- $m \in M$,
- clients$(m) \subset M$ is the set of modules that directly depend on module m,
- $|$clients$(m)|$ is the cardinality of clients(m).

Clustering Guideline: *Often, a set of omnipresent modules, such as a library, are grouped into a single subsystem.*

Central,Fringe,Omnipresent

Module Example

$$T_c = 8 \text{ and } T_f = 7 \text{ and } T_{op} = 3$$



Central, Fringe, and Omnipresent Modules

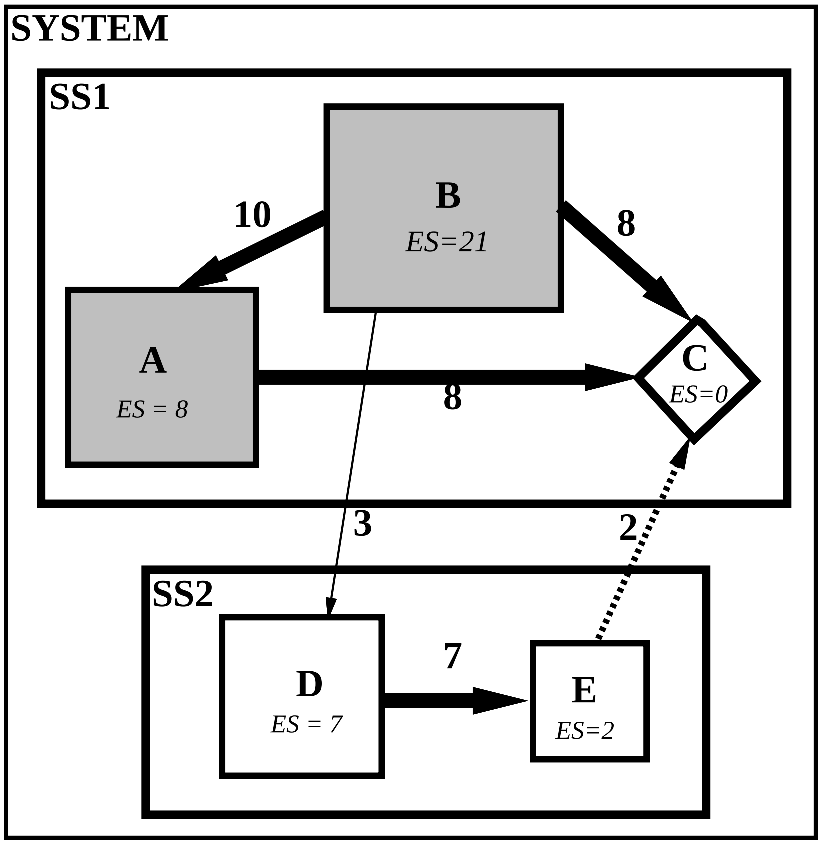
Large and Dark Nodes: Central Modules

Small and Light Nodes: Fringe Modules

Diamond Nodes: Omnipresent Modules

Subsystem Clustering

Related modules are clustered into subsystems using a special-purpose *Subsystem Interconnection Language (SIL).*



Subsystem Clustering

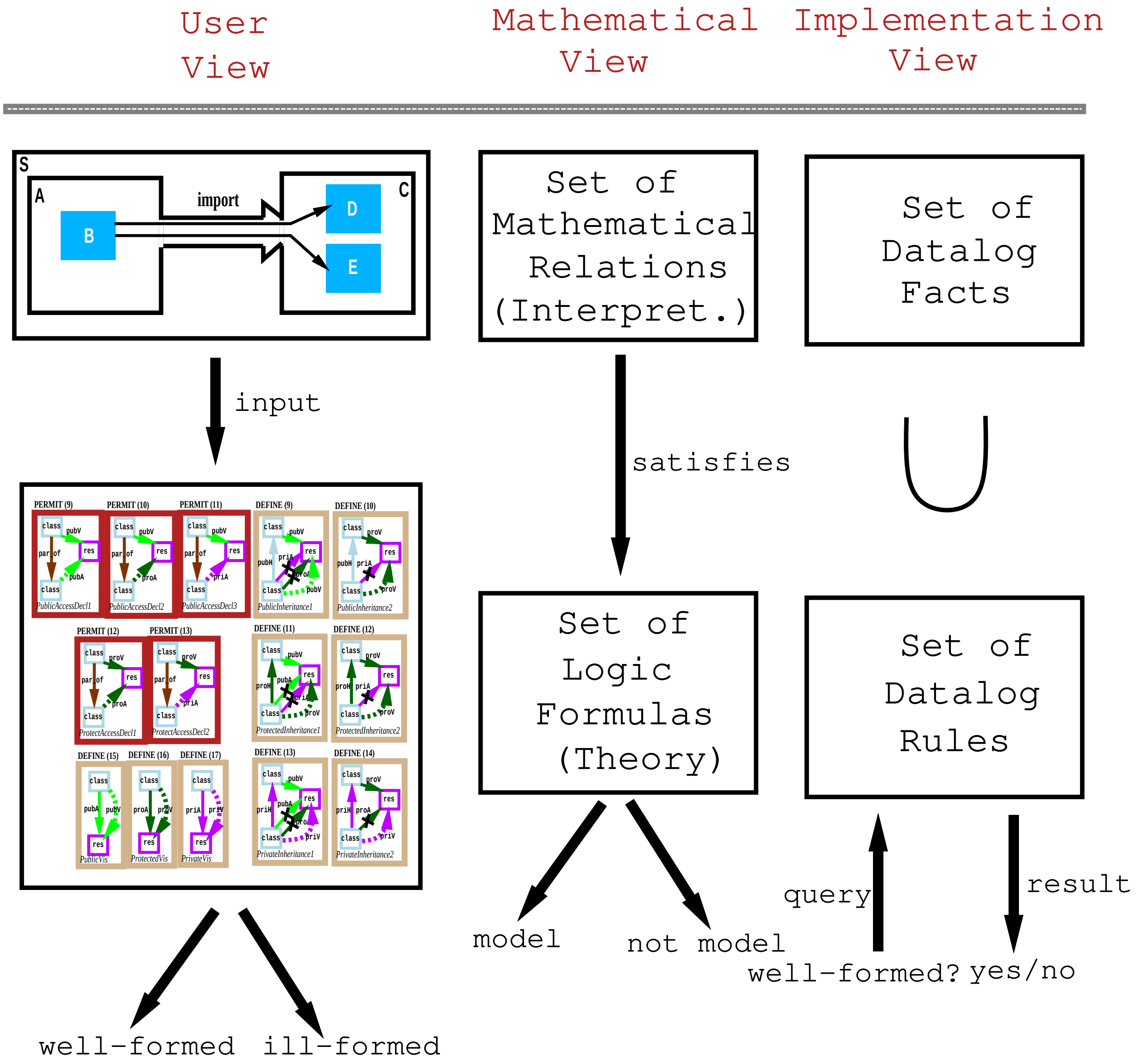
subsystem SYSTEM
contain SS1, SS2
end SYSTEM

subsystem SS1
contain A, B, C
end SS1

subsystem SS2
contain D, E
end SS2

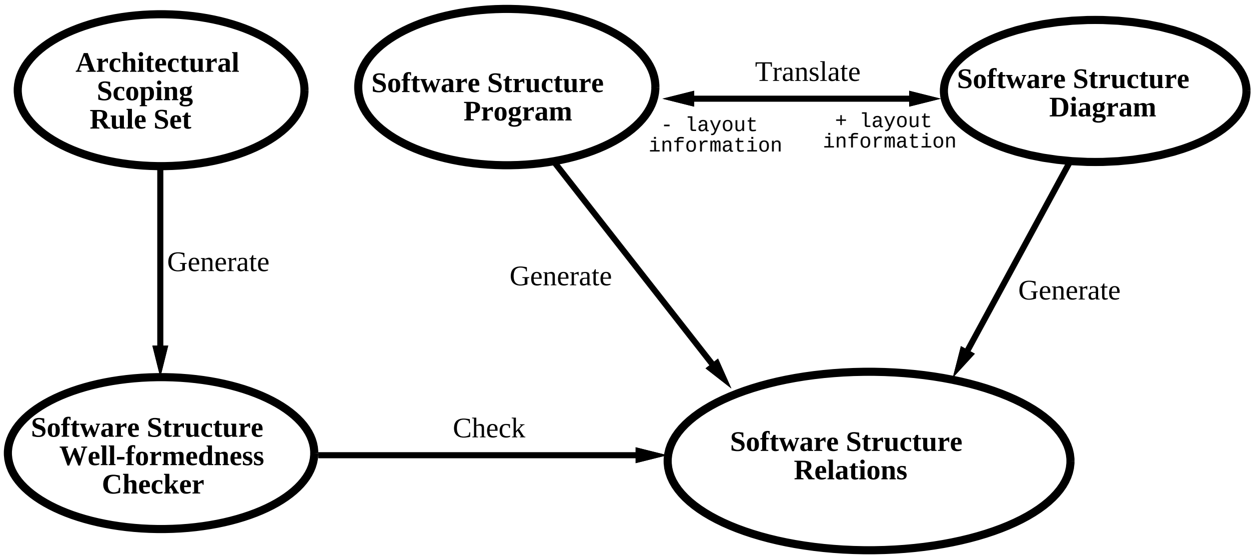
ASF Formalism

Conceptual Framework



[Mancoridis, 1996a]

Implementation Framework



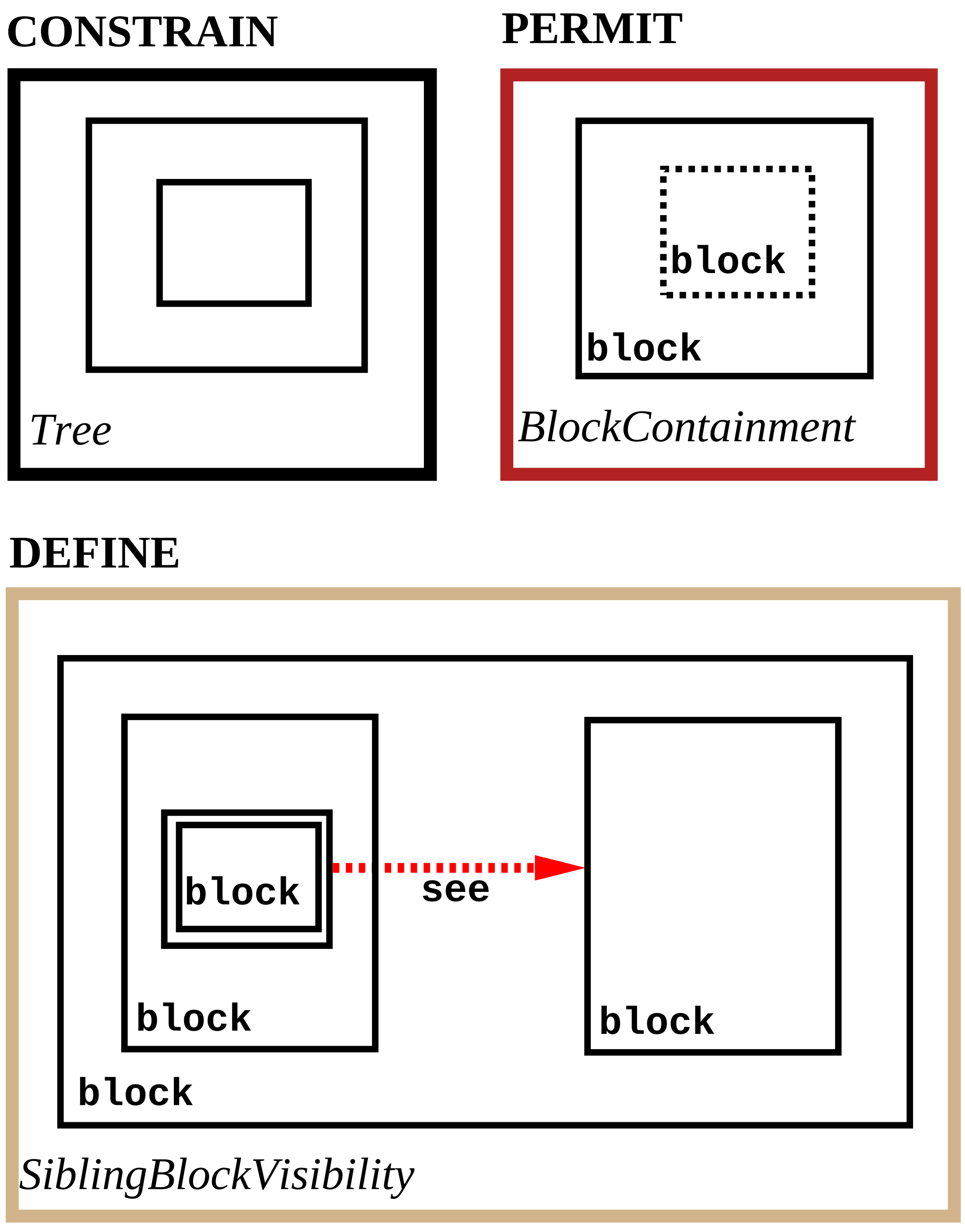
[Mancoridis and Holt, 1995]

Algol-60 Blocks Example

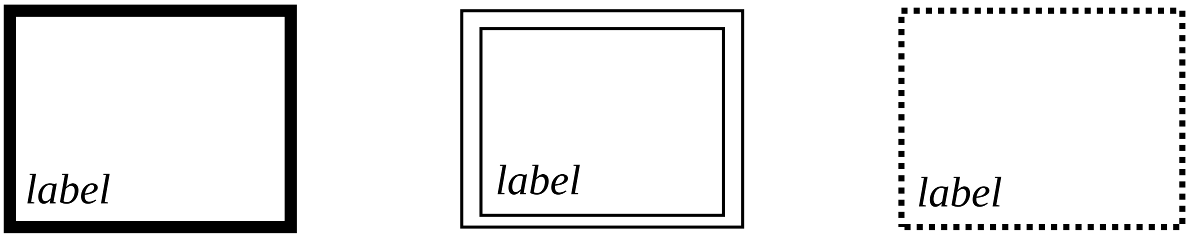
```
program MAIN;
-- local declarations for MAIN

    procedure P1;
     -- local declarations for P1

        procedure P3;
         -- local decalartions for P3                 Block
                                                      P3
         begin
             -- statements for P3
         end P3;
                                                               Block
                                                               P1
         procedure P4;
         -- local declarations for P4                                  Outer
                                                      Block           Block
         begin                                        P4              MAIN
             -- statements for P4
         end P4;

     begin
         -- statements for P1
     end P1;

    procedure P2;
     -- local declarations for P2
                                                               Block
    begin                                                      P2
         -- statements for P2
    end P2;

begin
    -- statements for MAIN
end MAIN.
```

[Naur, 1963]

Algol-60 Blocks Style

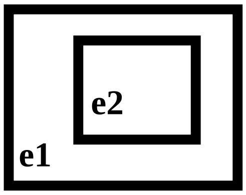


ASF Entities

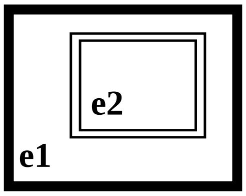


(simple) **(reflexiveTransitive)** **(permission)**

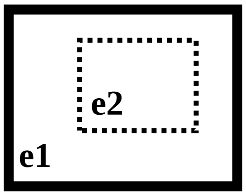
ASF Containment Relations



1. Entity contains simple entity

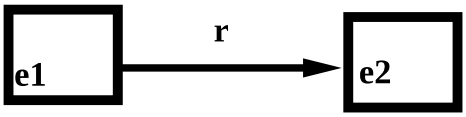


2. Entity contains transitive entity

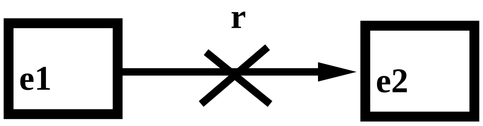


3. Entity contains permission entity

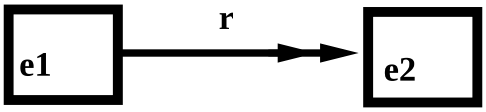
ASF Edge Relations



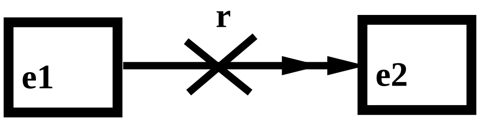
1. Simple Relation



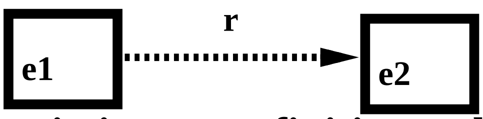
2. Negated Simple Relation



3. Reflexive Transitive Relation

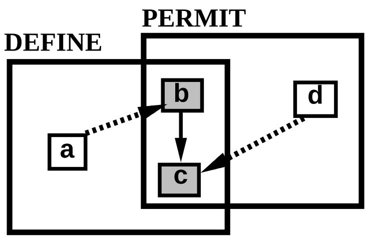


4. Negated Reflexive Transitive Relation

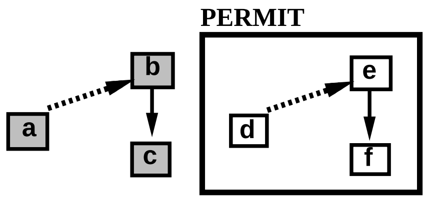


5, 6. Permission or Definition Relation

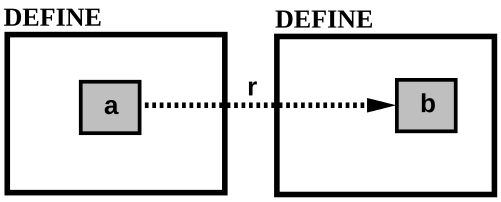
Syntactically Illegal ASF Rules



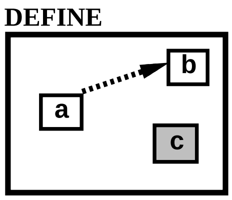
1. Rules cannot share entities and relations



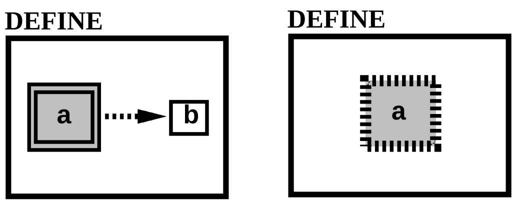
2. Entities and relations must belong to a rule



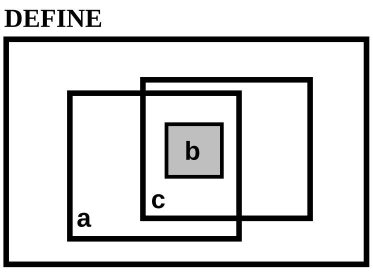
3. Relations must be between entities of the same rule



4. All entities must partake in a relation



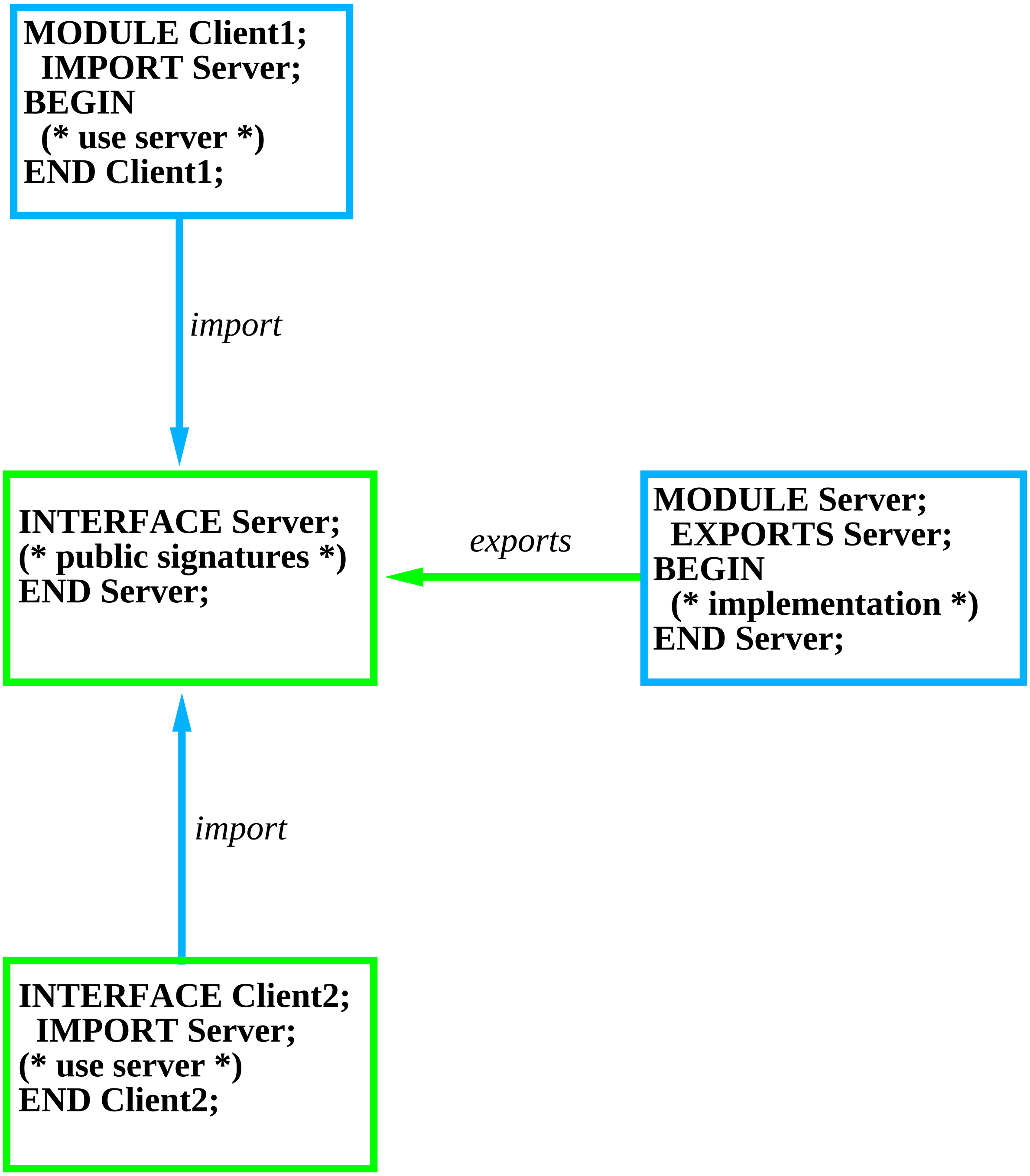
5. Double framed and dashed entities must be contained in an entity



6. An entity can only be contained in exactly one entity

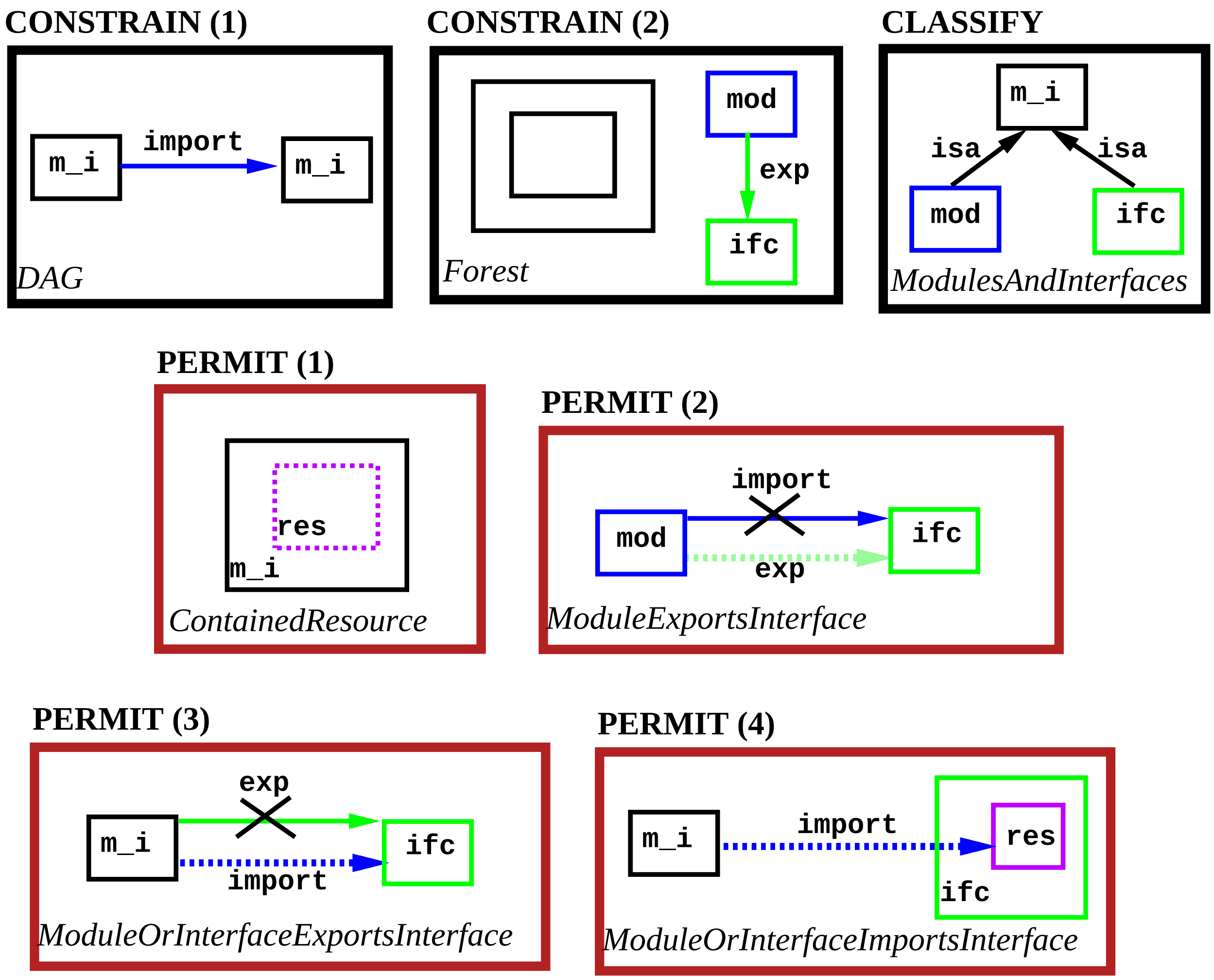
Programming Language Support for Modularity and Information Hiding

Modula-3 Example



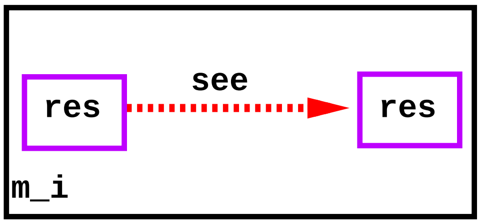
[Cardelli et al., 1992]

Modula-3 Style (Part 1)



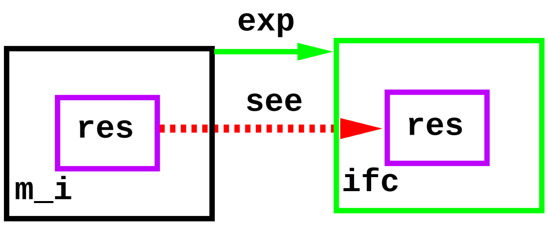
Modula-3 Style (Part 2)

DEFINE (1)



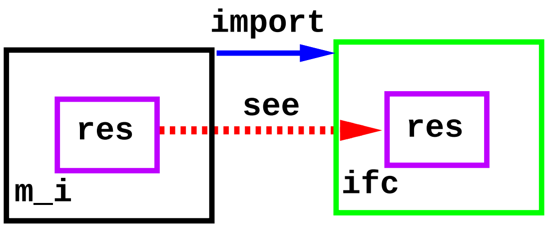
VisibilityBetweenSiblingResources

DEFINE (2)



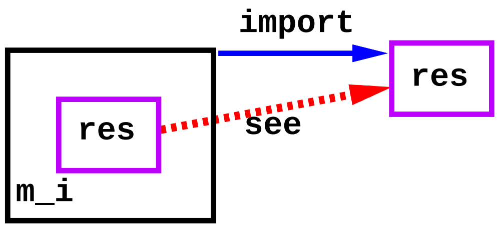
ExportVisibility

DEFINE (3)



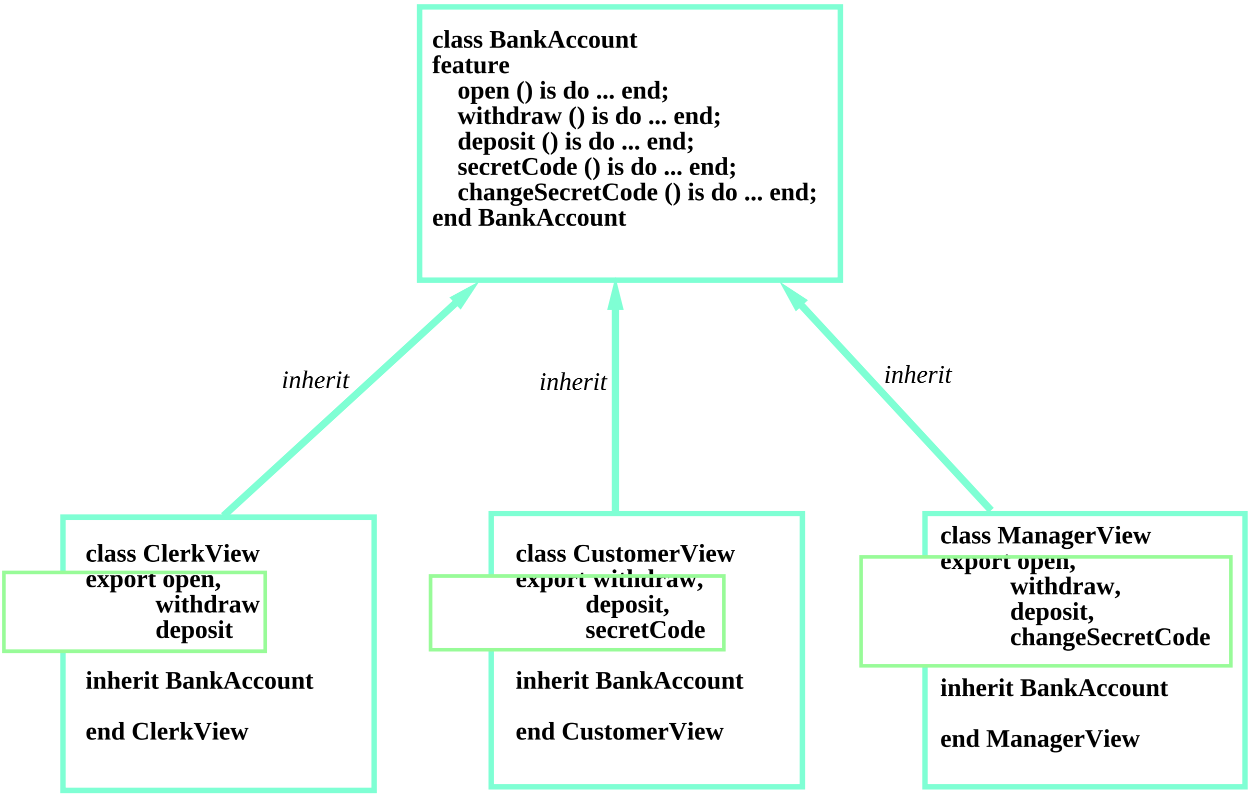
ImportVisibility

DEFINE (4)



SelectiveImportVisibility

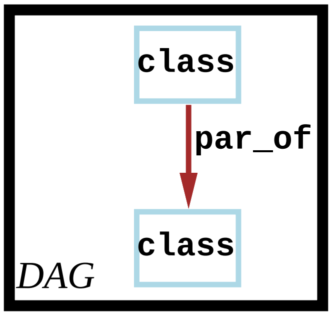
Eiffel Example



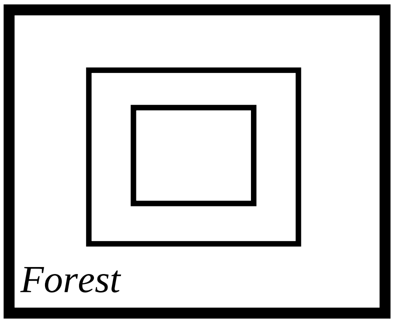
[Meyer, 1988]

Eiffel Style (Part 1)

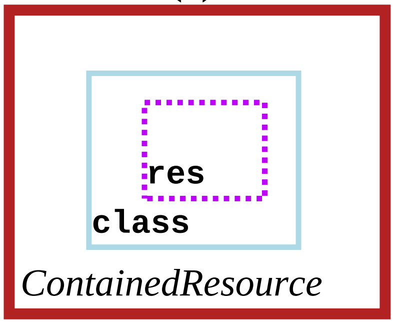
CONSTRAIN (1)



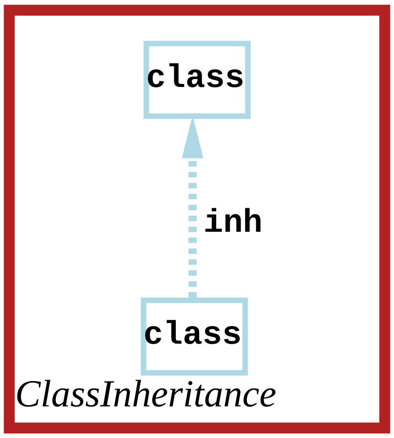
CONSTRAIN (2)



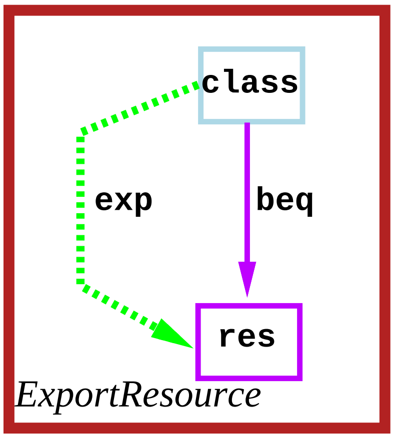
PERMIT (1)



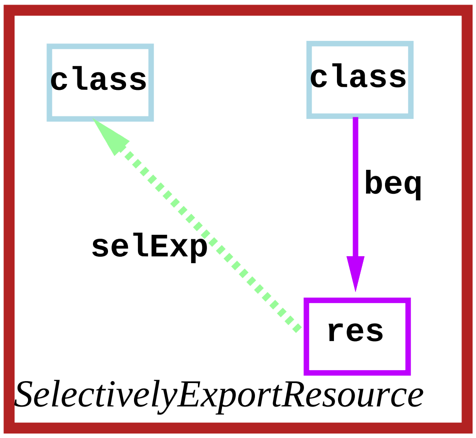
PERMIT (2)



PERMIT (3)

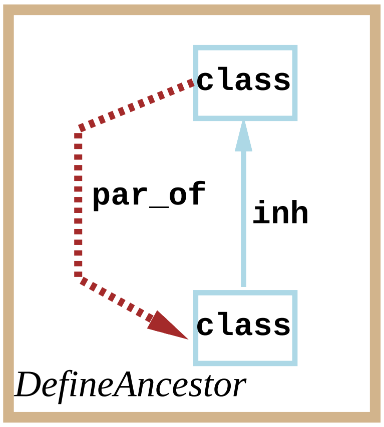


PERMIT (4)



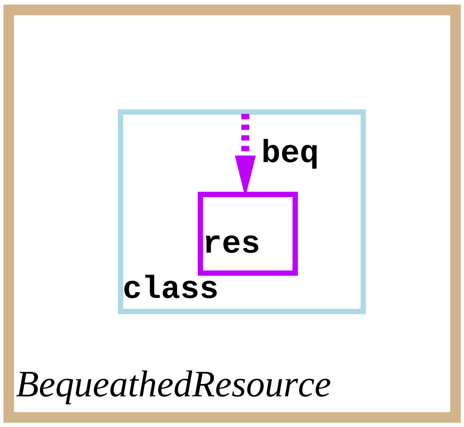
Eiffel Style (Part 2)

DEFINE (1)



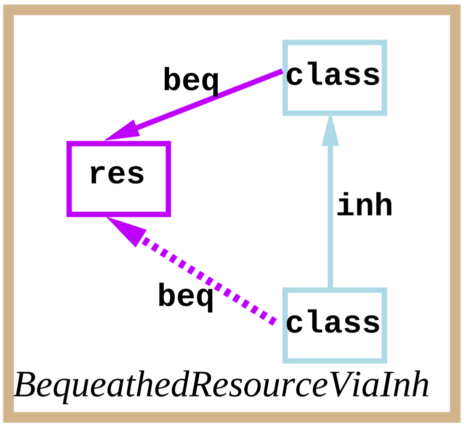
DefineAncestor

DEFINE (2)



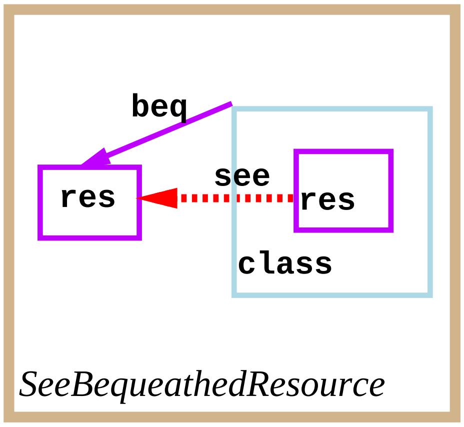
BequeathedResource

DEFINE (3)



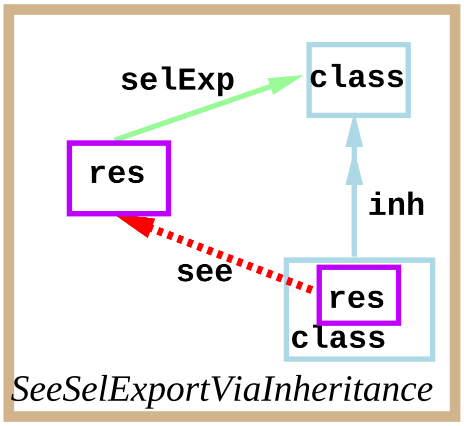
BequeathedResourceViaInh

DEFINE (4)



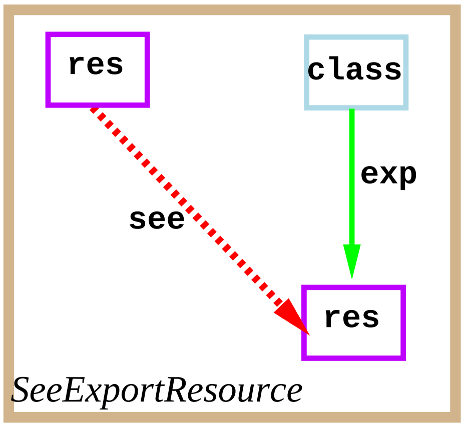
SeeBequeathedResource

DEFINE (5)



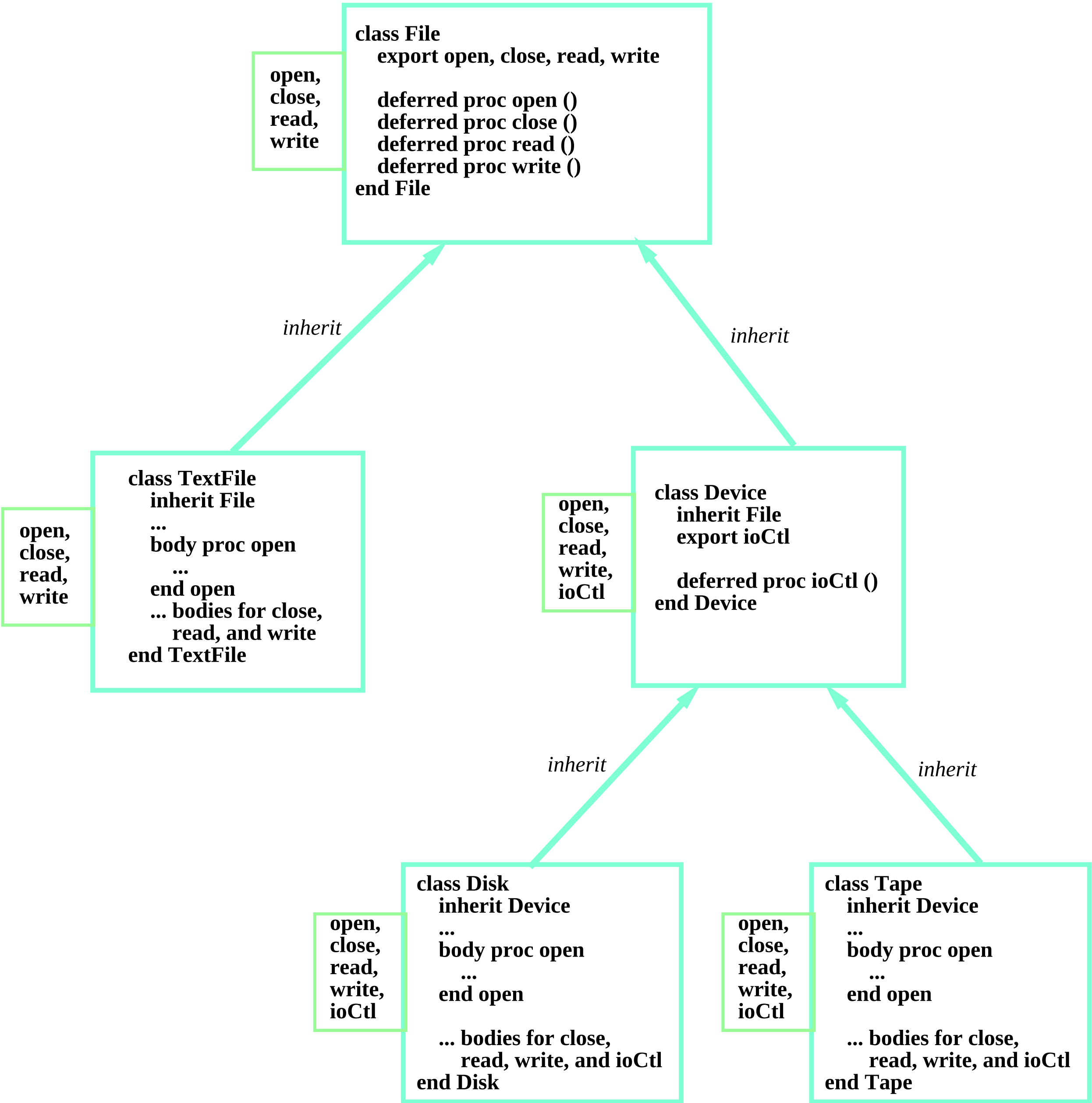
SeeSelExportViaInheritance

DEFINE (6)



SeeExportResource

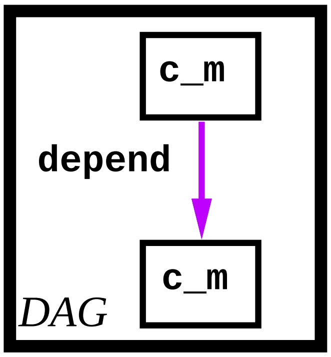
OOT Example



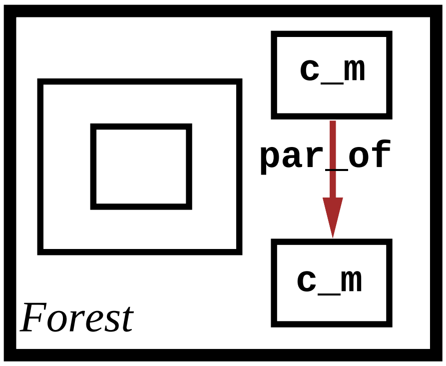
[Holt and West, 1994]

OOT Style (Part 1)

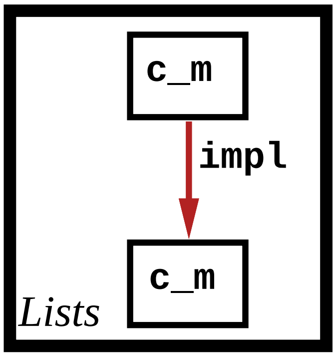
CONSTRAIN (1)



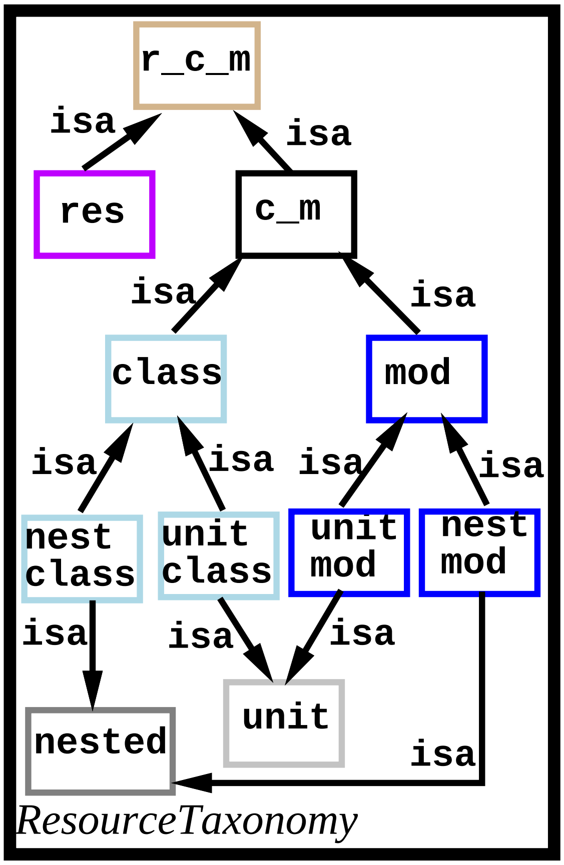
CONSTRAIN (2)



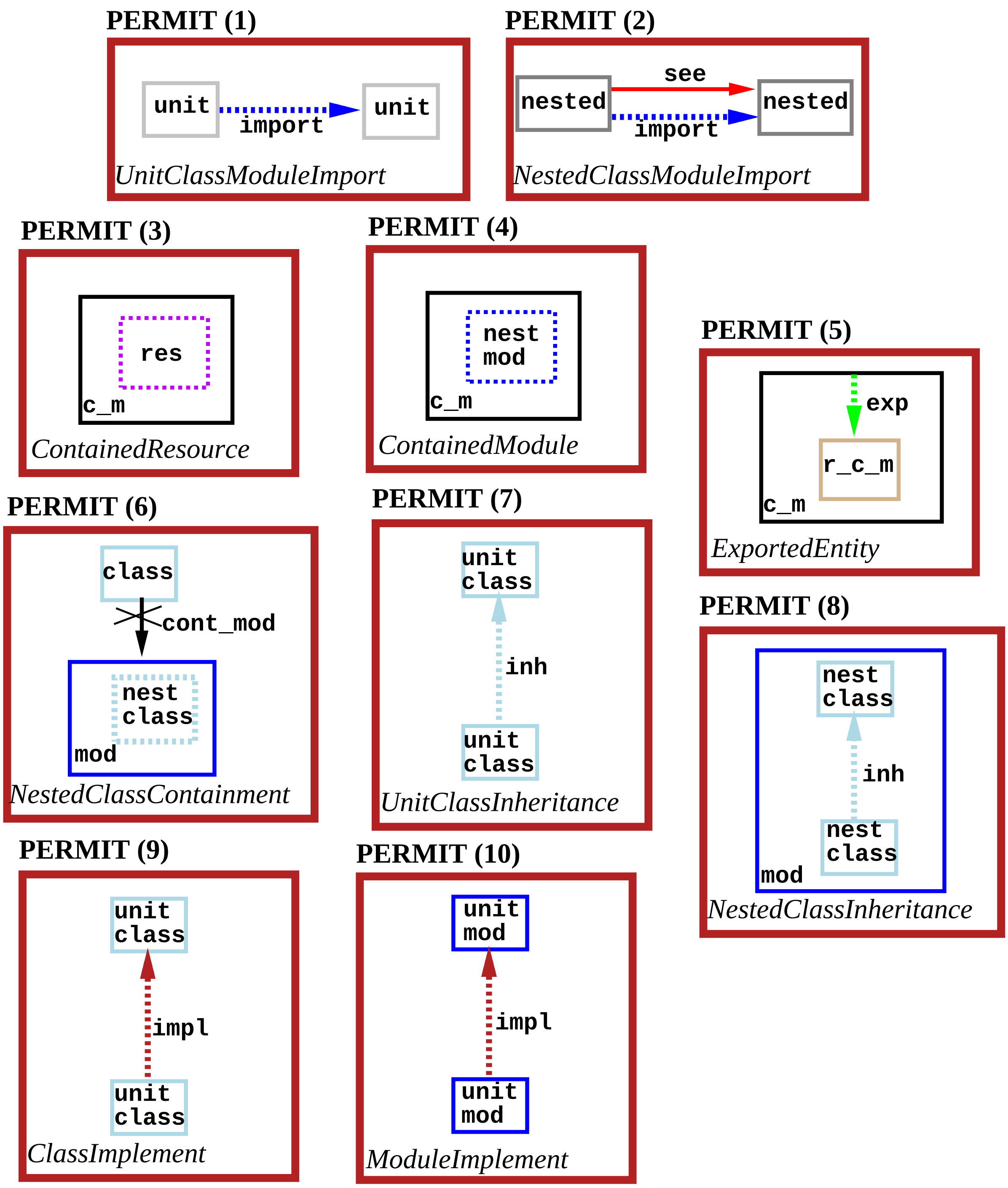
CONSTRAIN (3)



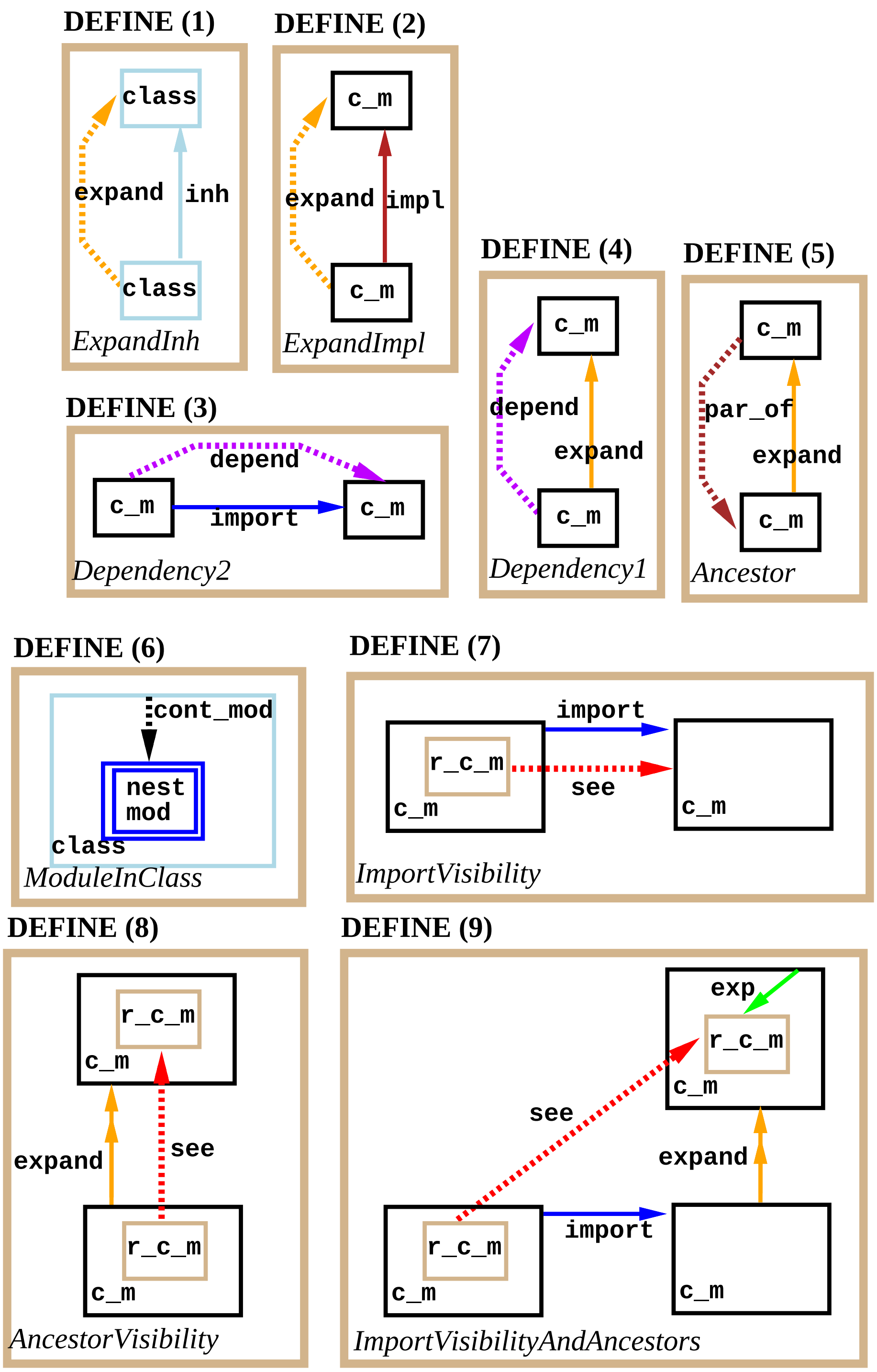
CLASSIFY



OOT Style (Part 2)



OOT Style (Part 3)



C++ Example

```
class Stack {
public:
    virtual void push (char *);
    virtual char * pop ();
    stack ();     // constructor
    ~stack ();   // destructor
protected:
    const maxdepth = 100;
    int top;
    char * contents [maxdepth];
};

void stack :: push (char *s) {
    contents [++top] = s;
}

... bodies for pop, stack, and  ~stack.
```

publically inherit

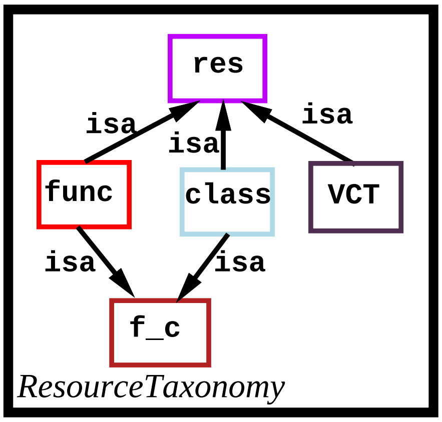
```
class ExtendStack : public Stack {
public:
    inline int depth () {
        return top;
    }
    inline void push (char *s) {
        pushCount += 1;
        Stack :: push (s);
    }
protected:
    int pushCount = 0;
};
```

also:
pop()

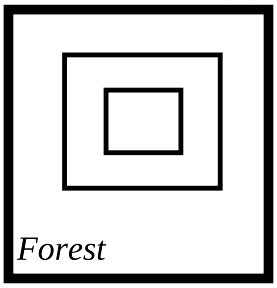
[Stroustrup, 1986]

C++ Style (Part 1)

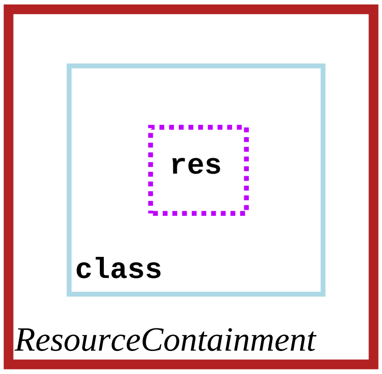
CLASSIFY



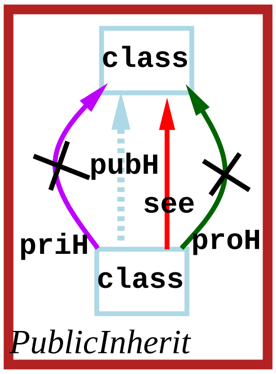
CONSTRAIN (1)



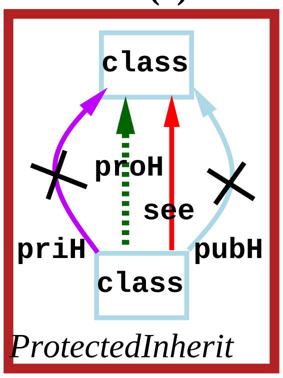
PERMIT (1)



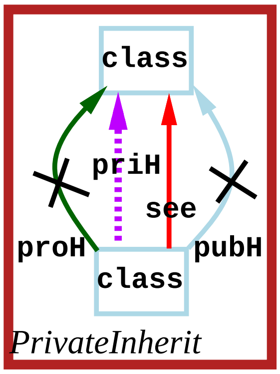
PERMIT (2)



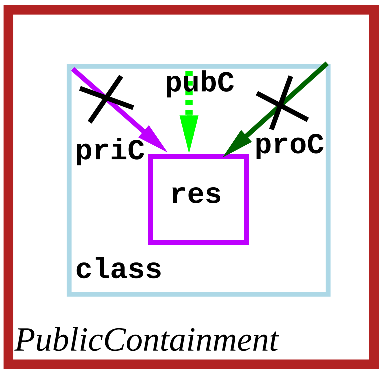
PERMIT (3)



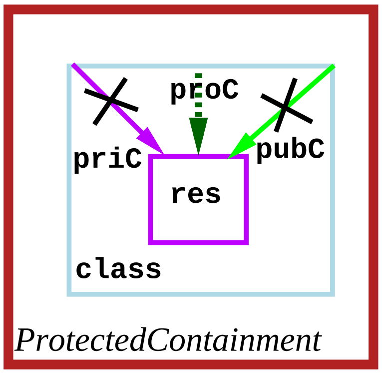
PERMIT (4)



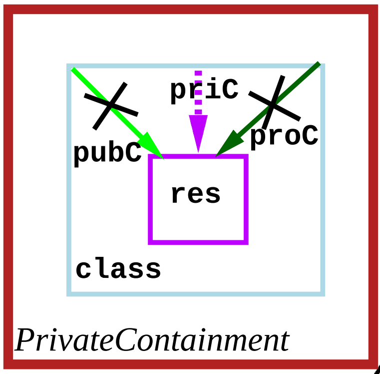
PERMIT (5)



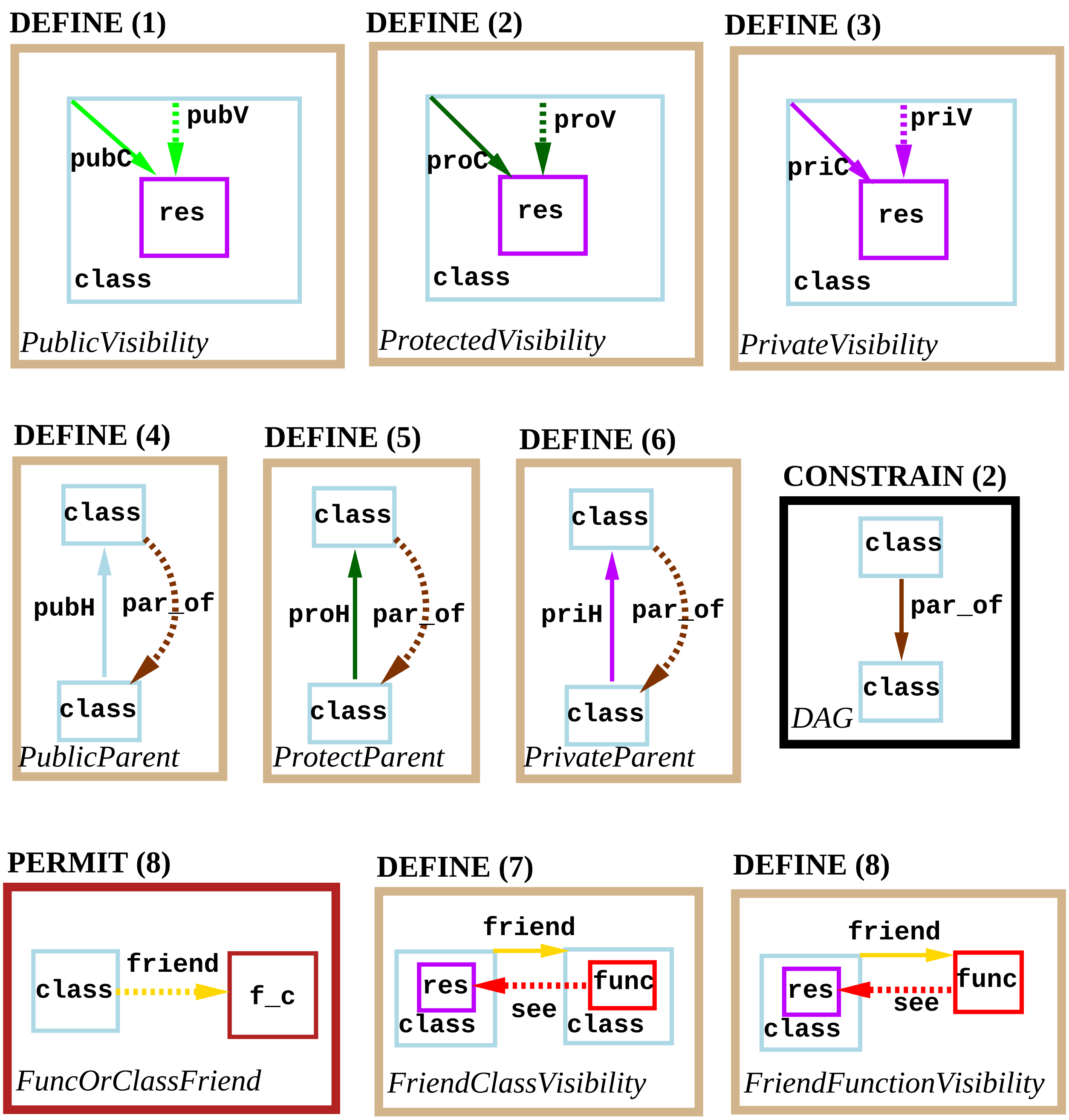
PERMIT (6)



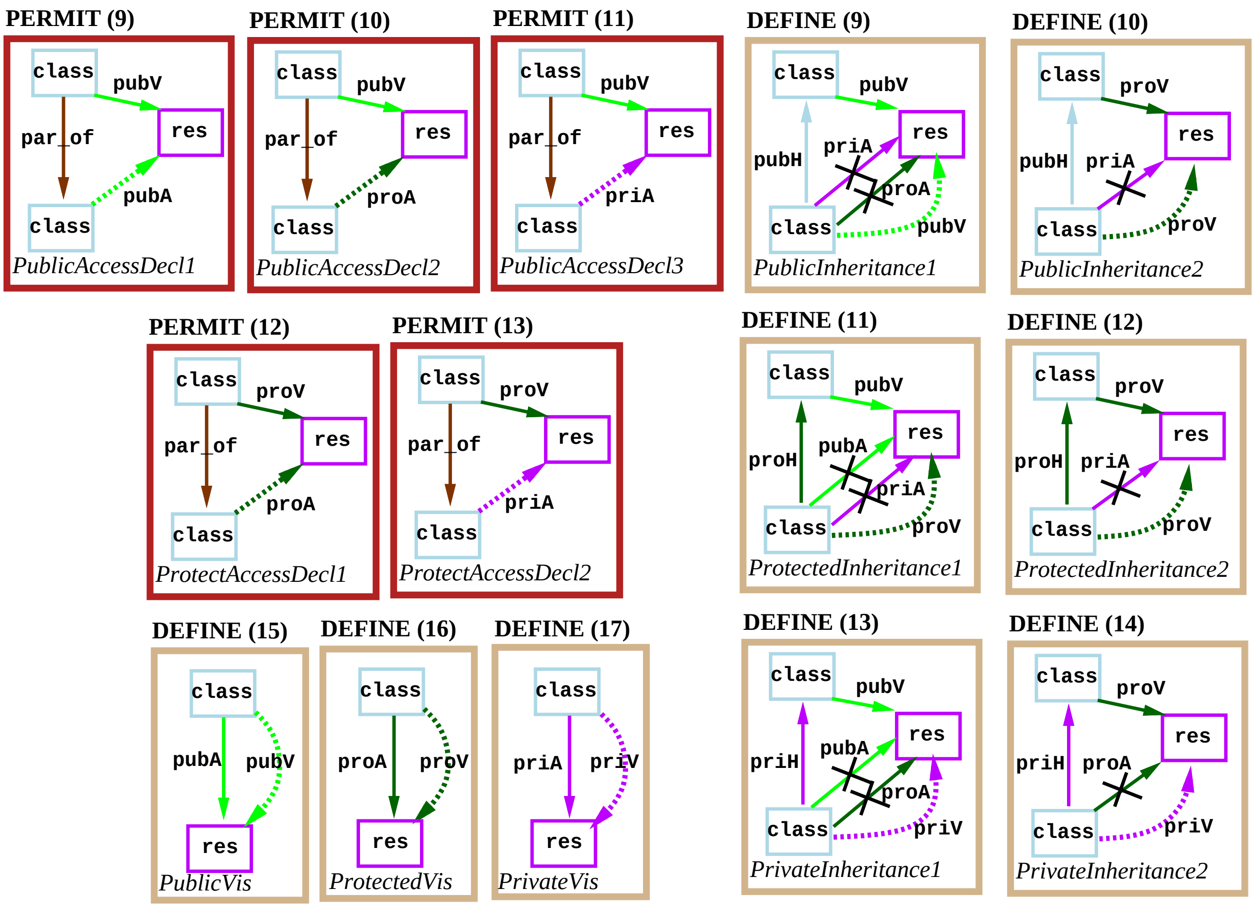
PERMIT (7)



C++ Style (Part 2)

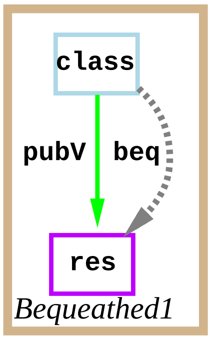


C++ Style (Part 3)



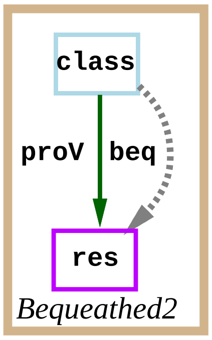
C++ Style (Part 4)

DEFINE (18)



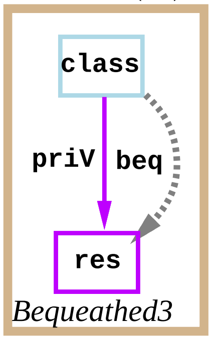
Bequeathed1

DEFINE (19)



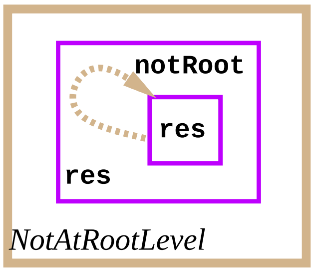
Bequeathed2

DEFINE (20)



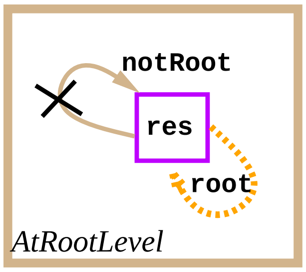
Bequeathed3

DEFINE (21)



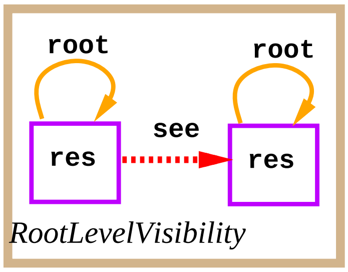
NotAtRootLevel

DEFINE (22)



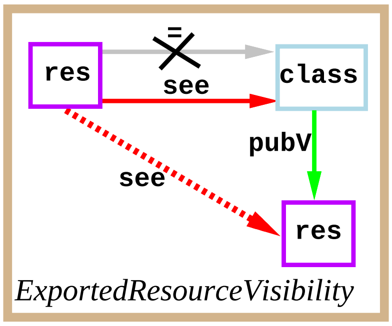
AtRootLevel

DEFINE (23)



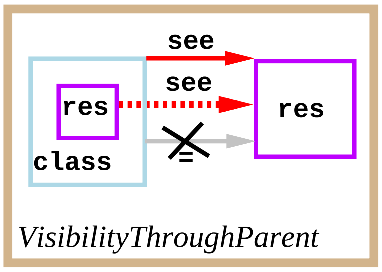
RootLevelVisibility

DEFINE (24)



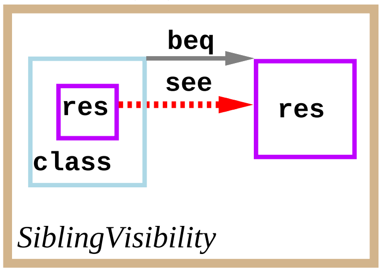
ExportedResourceVisibility

DEFINE (25)



VisibilityThroughParent

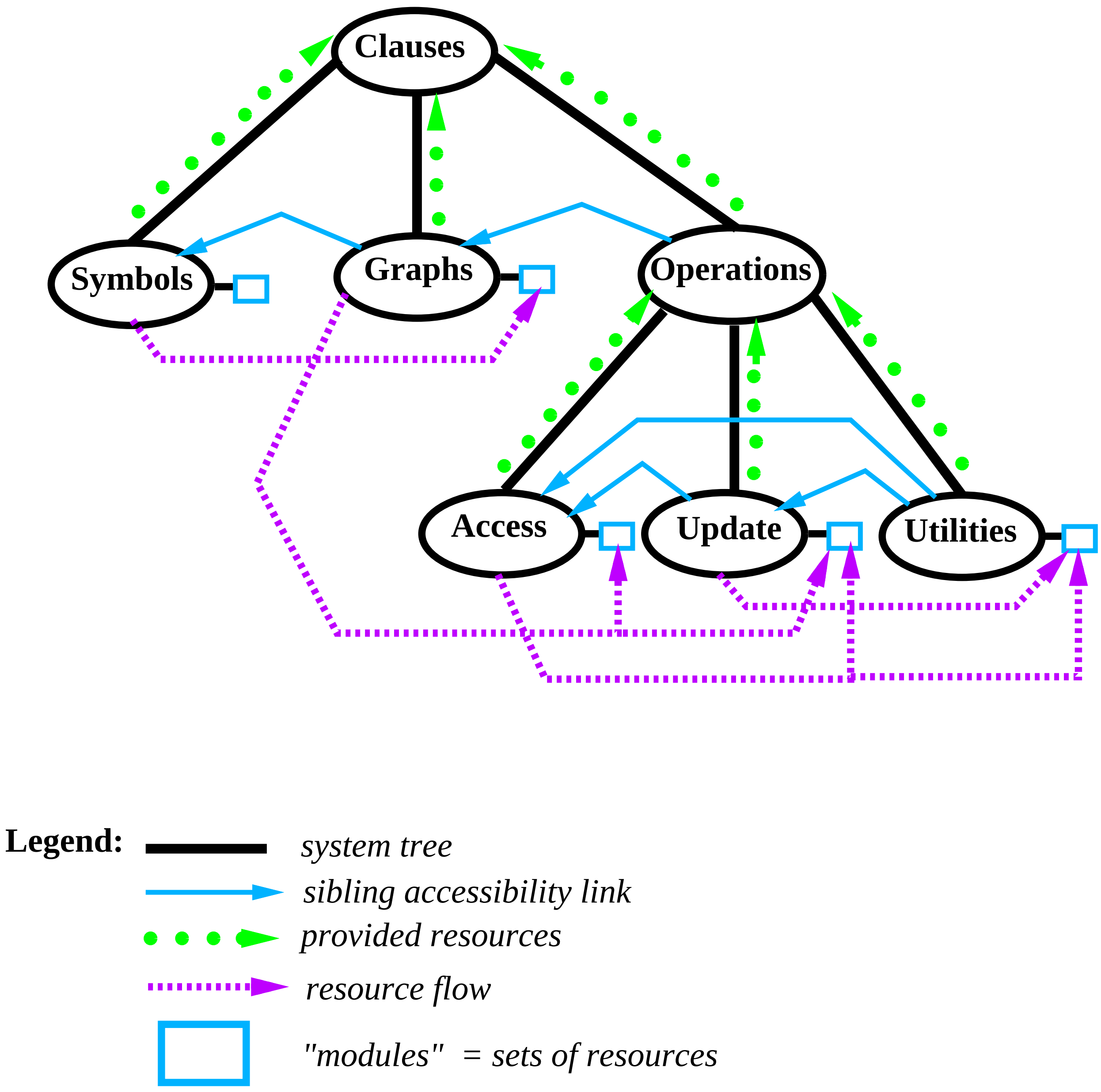
DEFINE (26)



SiblingVisibility

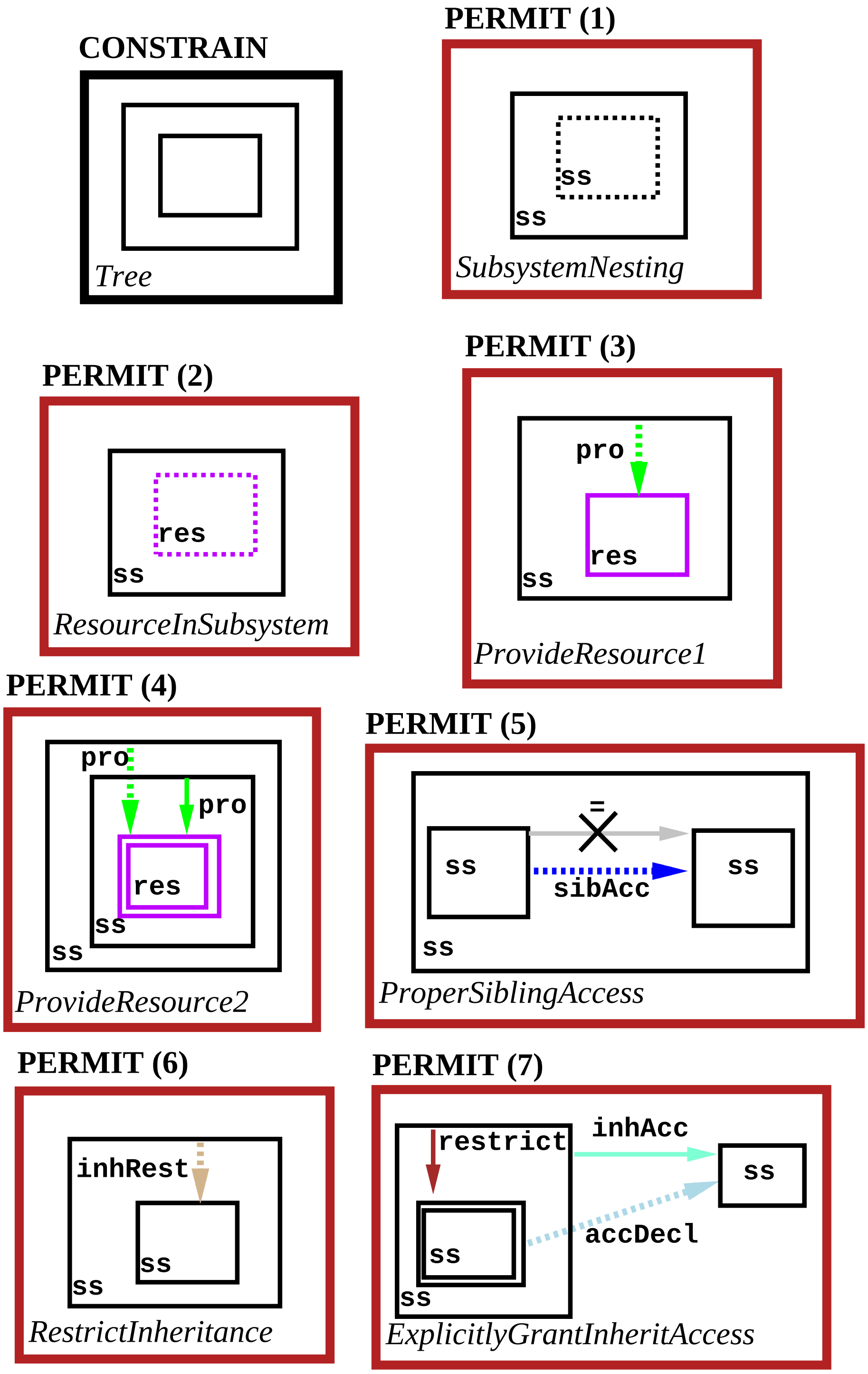
Module Interconnection Languages

MIL75 Example



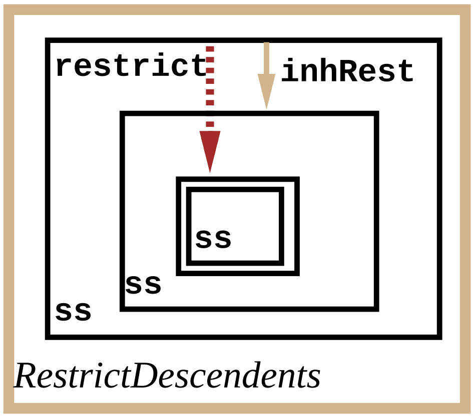
[DeRemer and Kron, 1976]

MIL75 Style (Part 1)



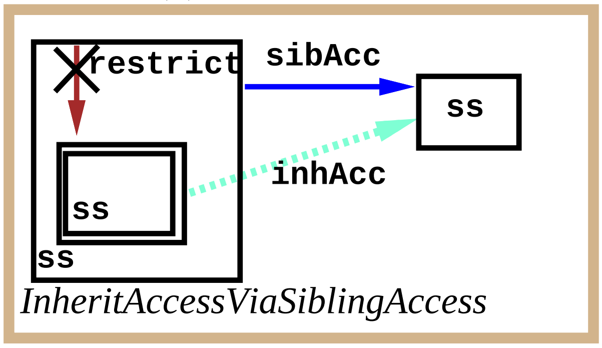
MIL75 Style (Part 2)

DEFINE (1)



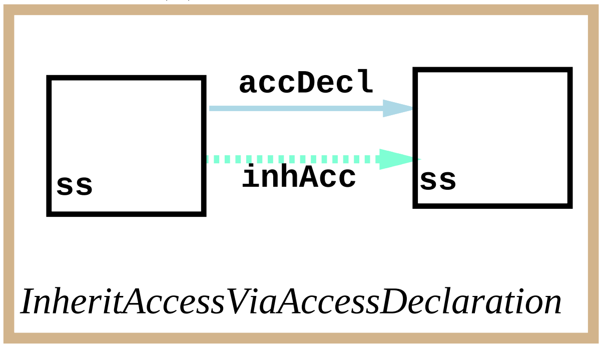
RestrictDescendents

DEFINE (2)



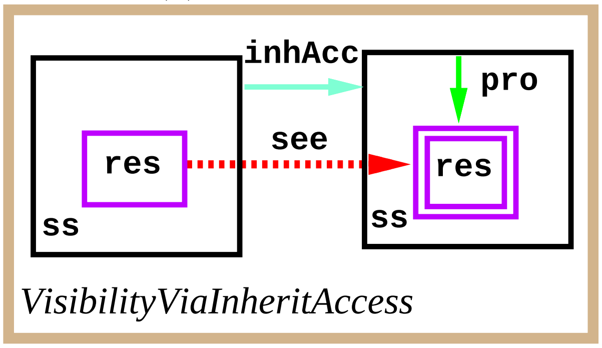
InheritAccessViaSiblingAccess

DEFINE (3)



InheritAccessViaAccessDeclaration

DEFINE (4)



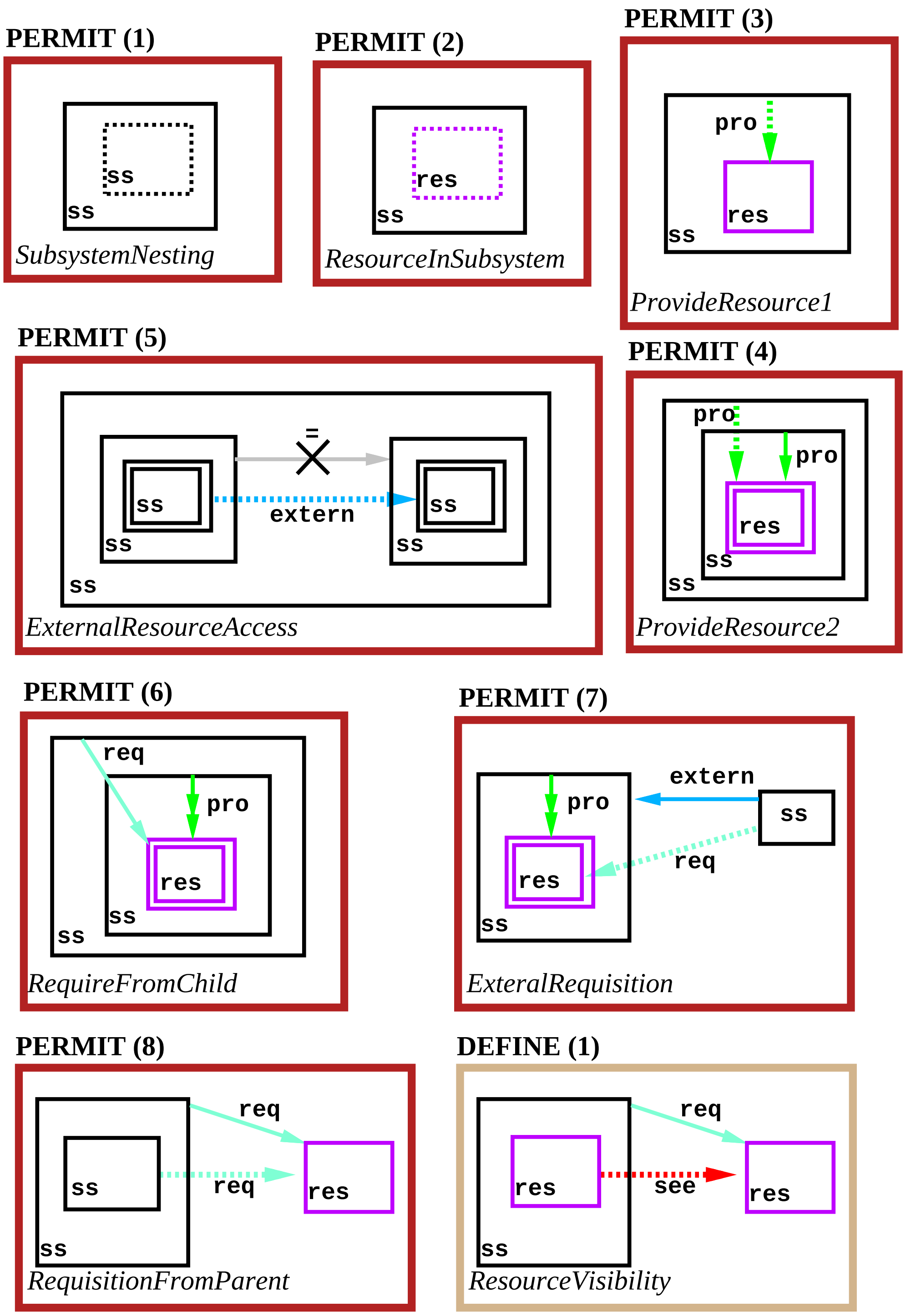
VisibilityViaInheritAccess

Cooprider MIL Example

```
subsystem S
   provides Useful1, Useful2, Other1
   requires SymbolTable, Other1, Other2, ListBasic external LIST

   subsystem ST
      provides SymbolTable
      ...
   end ST


   subsystem S1
      provides Other1
      requires ListBasic, SymbolTable
      ...
    end S1


   subsystem S2
     provides Other2
     requires ListBasic, SymbolTable, Useful1, Other1
     ...
   end S2

   ...
end S
```

[Cooprider, 1979]

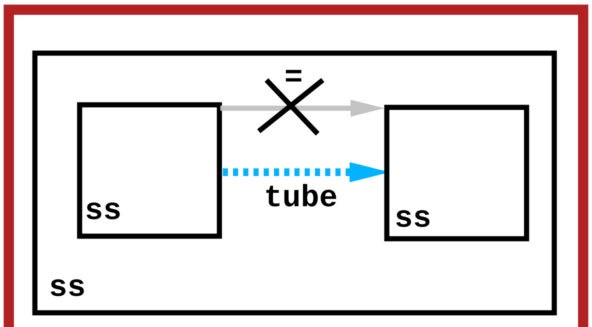
Cooprider MIL Style



Styles of Software Structure

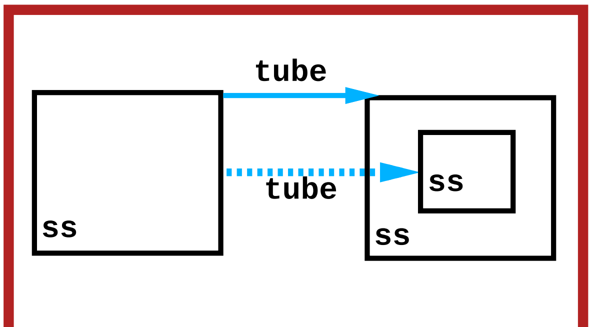
Tube Style

PERMIT



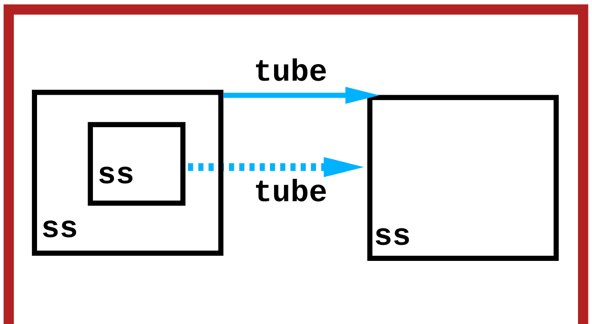
TubeBetweenProperSiblings

PERMIT



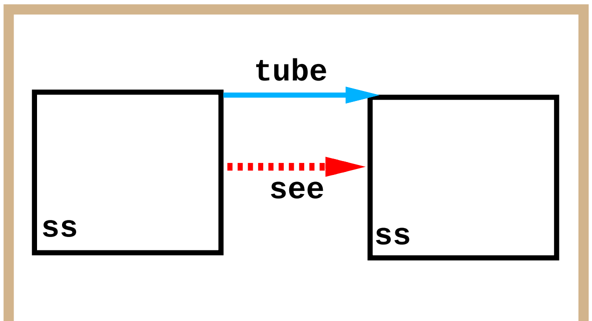
ChildTubeViaParentTube3

PERMIT



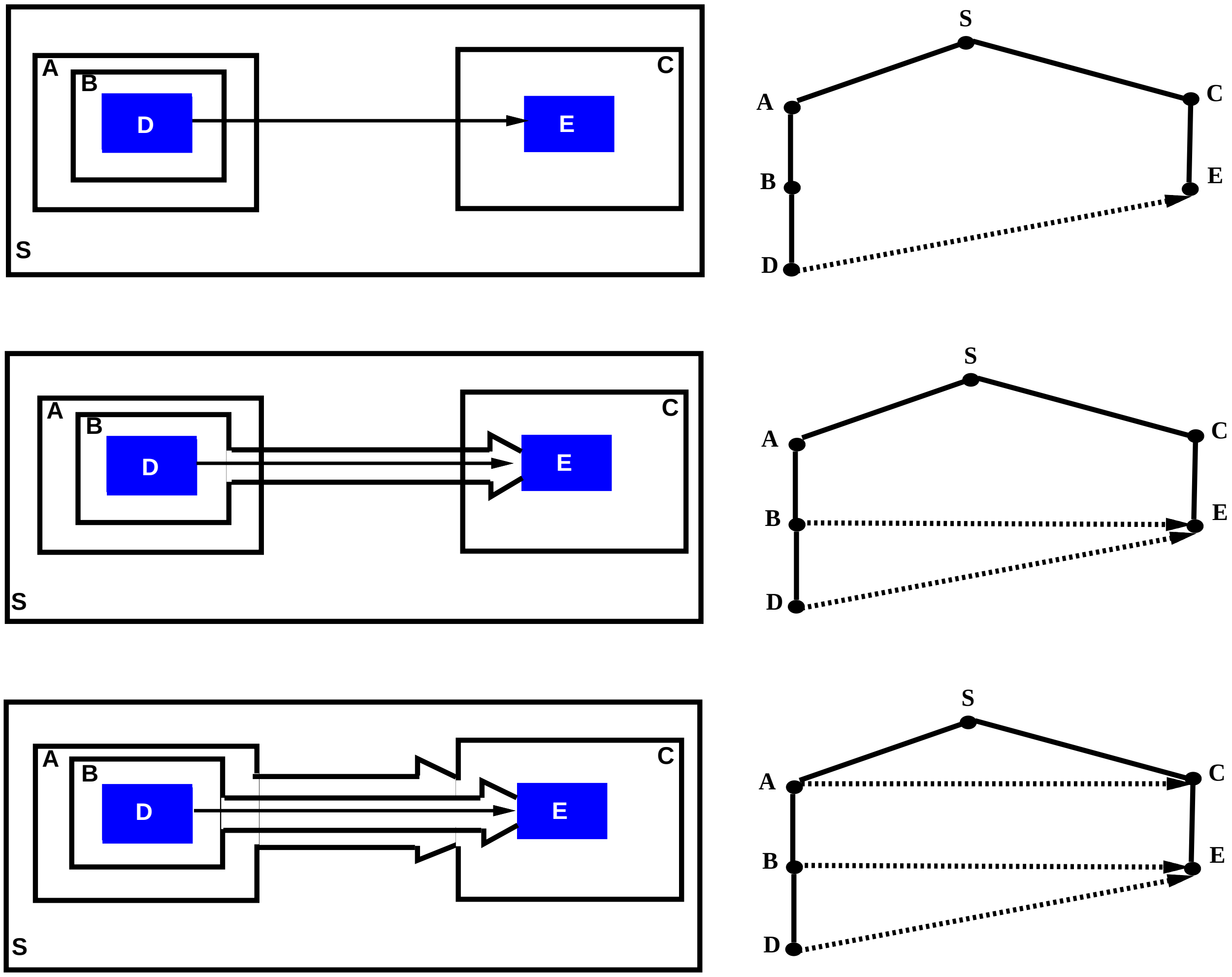
ChildTubeViaParentTube2

DEFINE



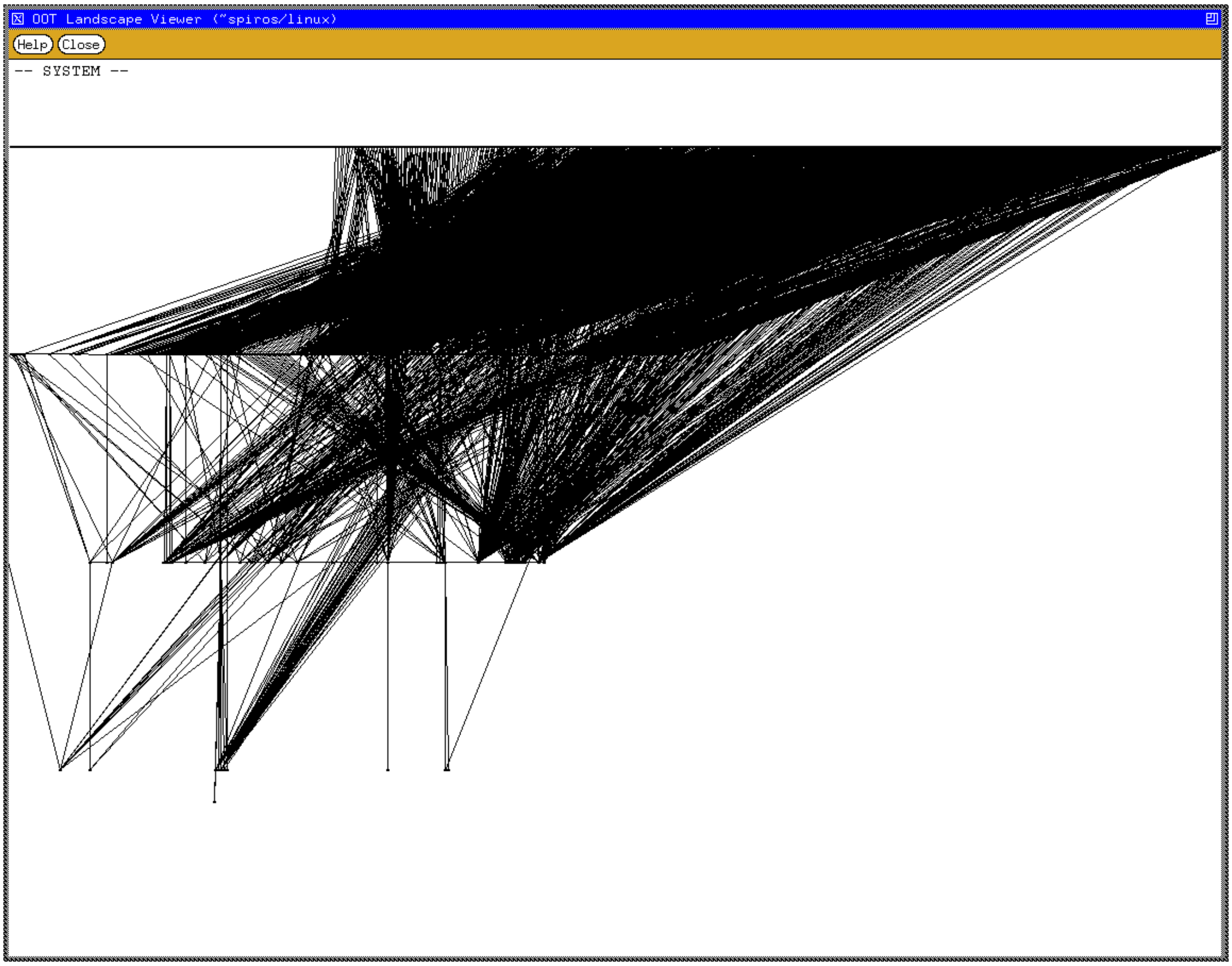
TubesImplyVisibility

Tube Algorithm

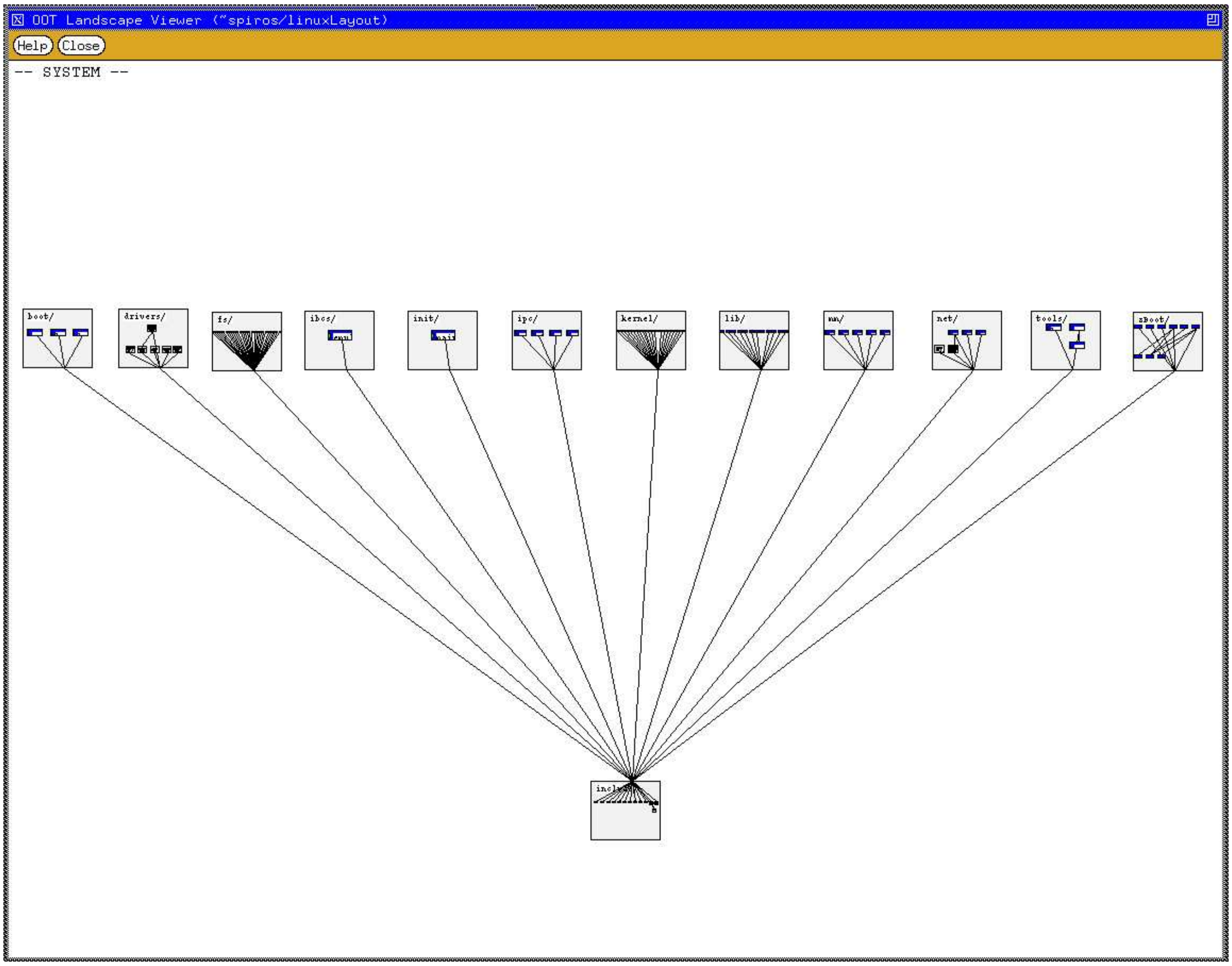


[Mancoridis and Holt, 1996]

Linux Structure

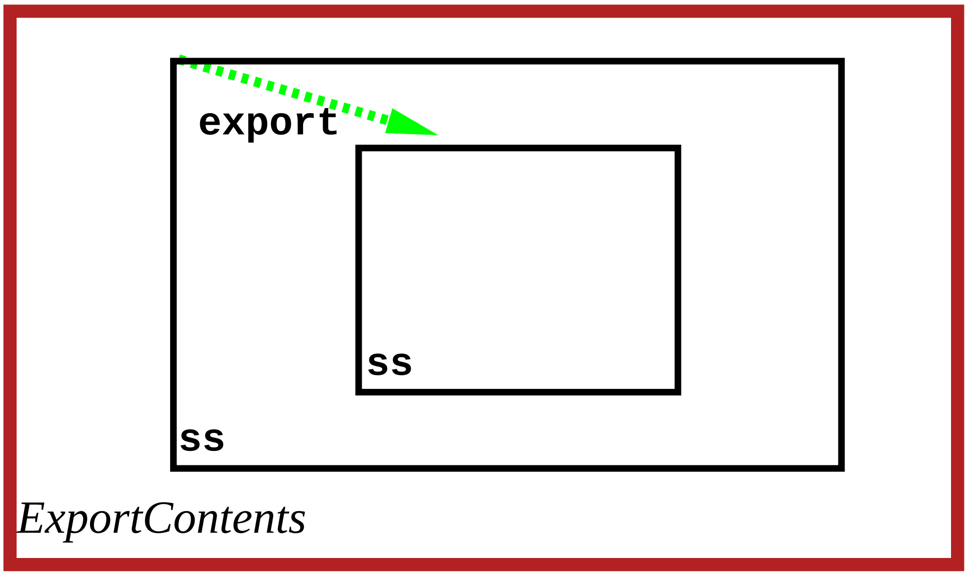


Revised Linux Structure

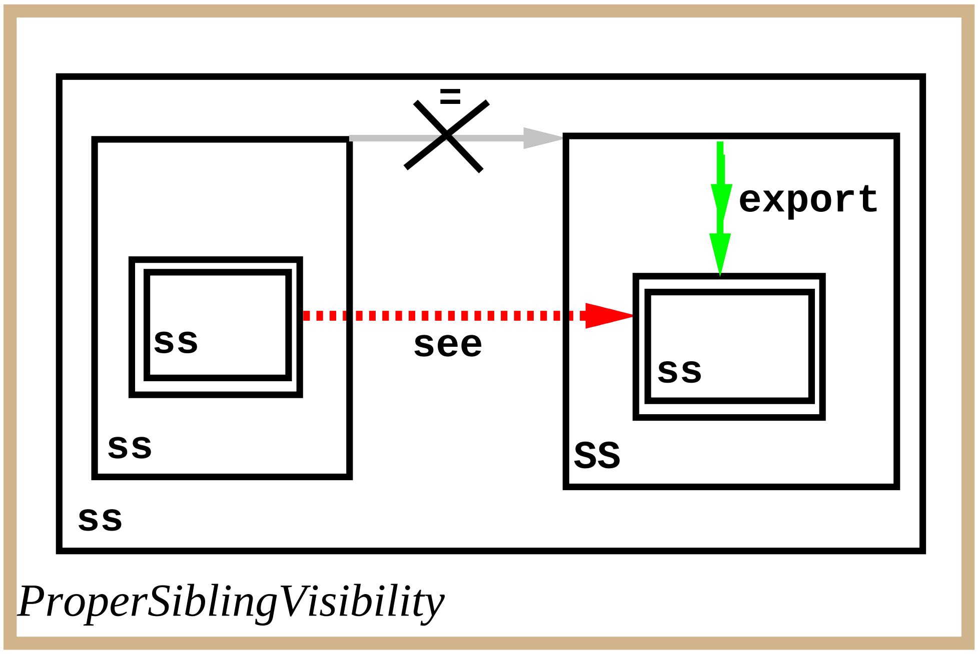


Export Style

PERMIT

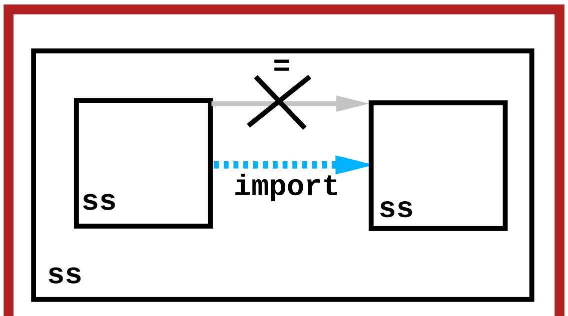


DEFINE



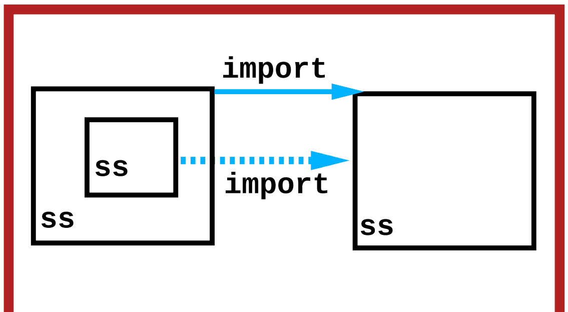
Import Style

PERMIT



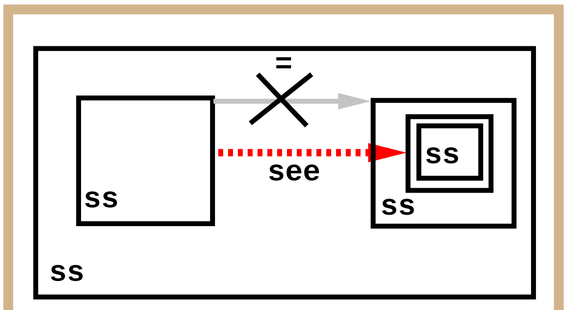
ProperSiblingImport

PERMIT



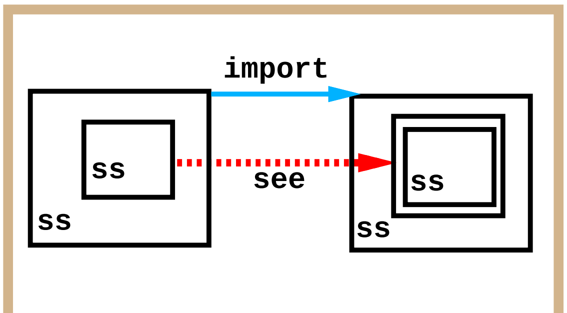
ChildImportsViaParentImport

DEFINE



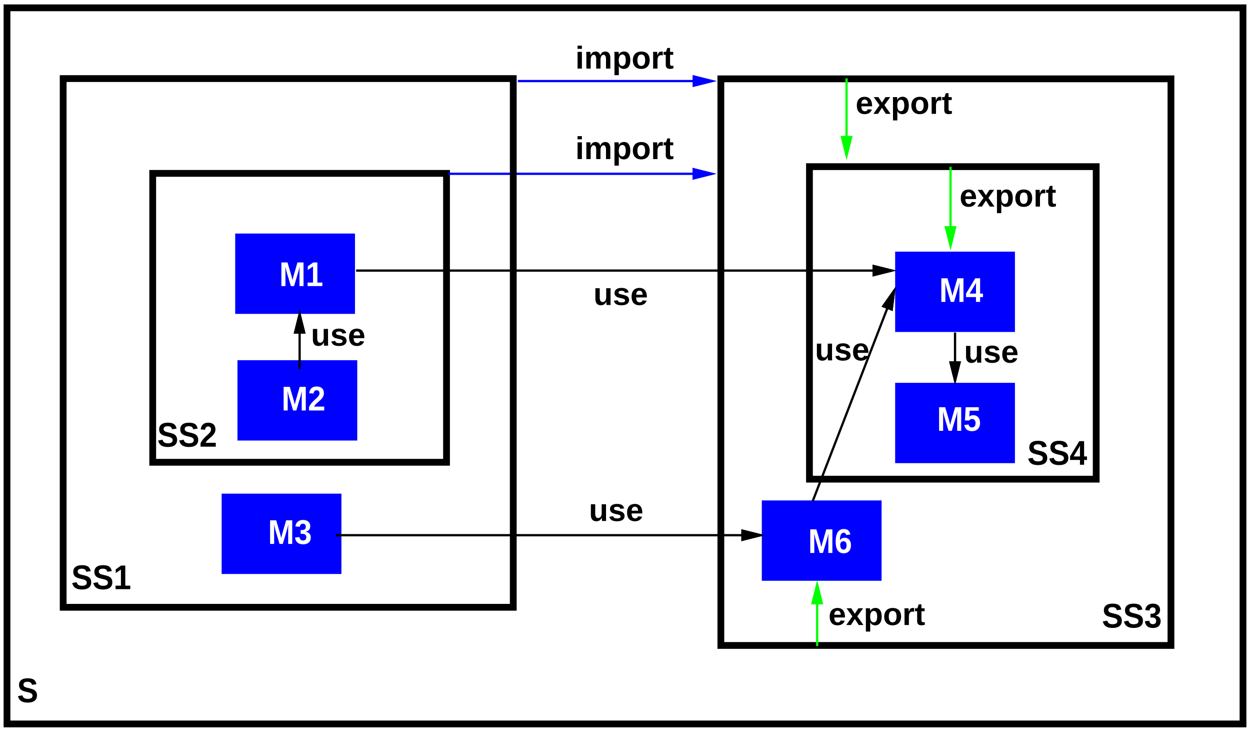
ProperSiblingVisibility

DEFINE

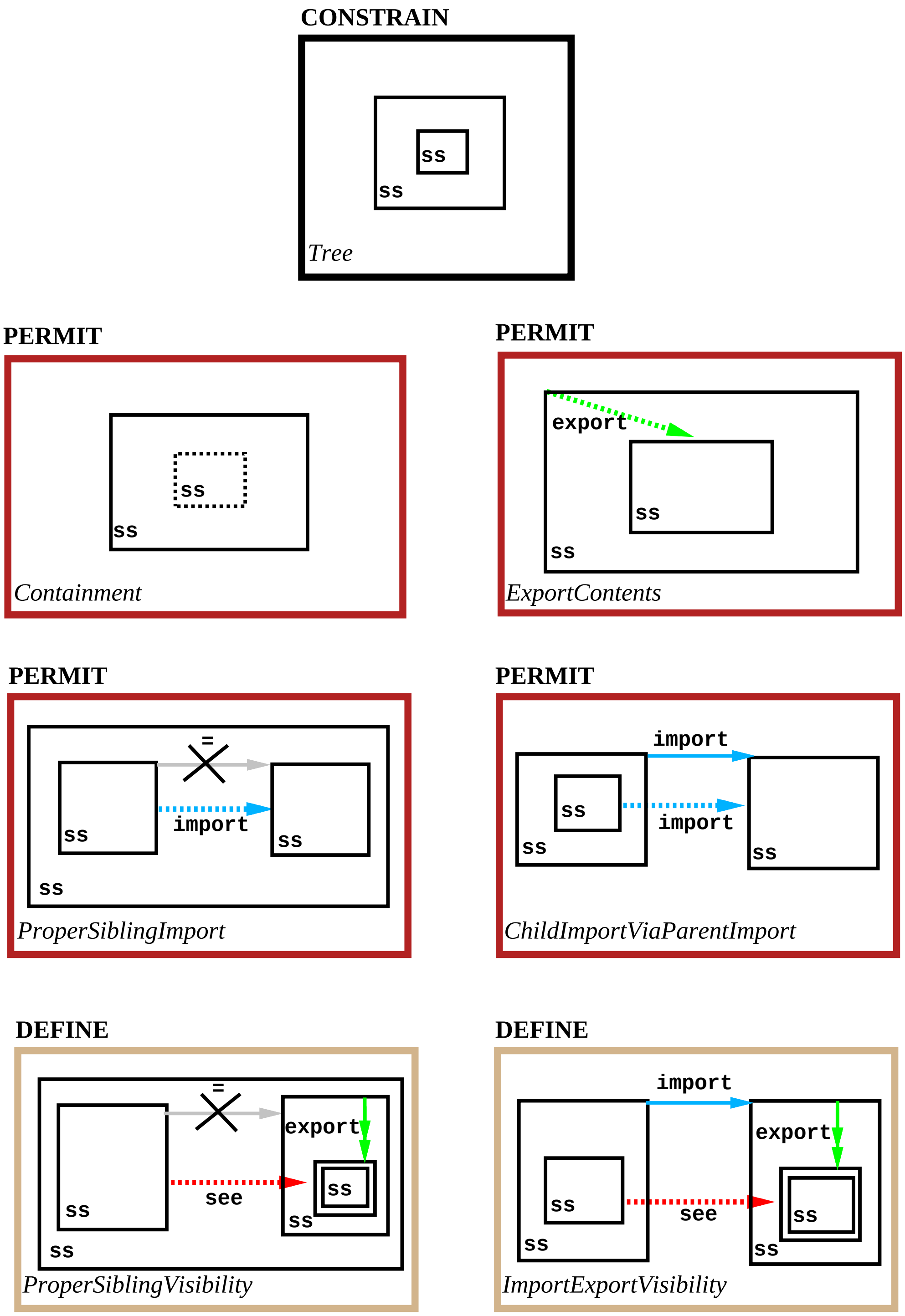


ImportVisibility

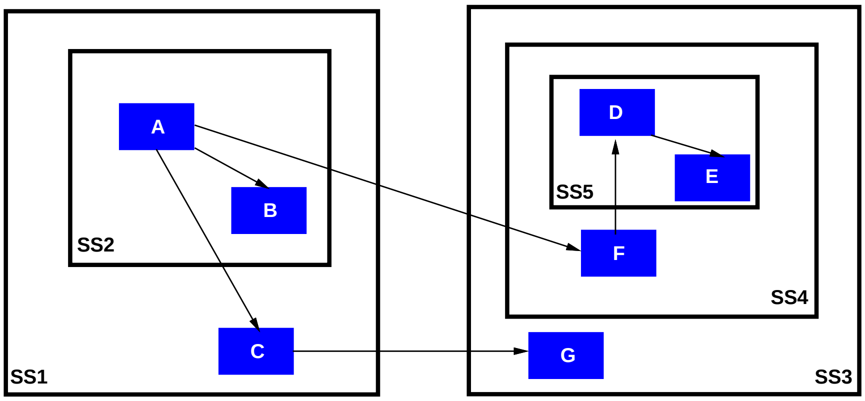
Import/Export Example



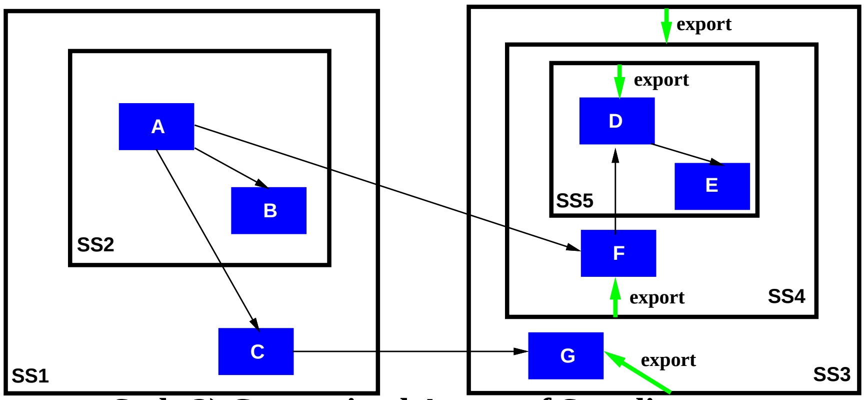
Import/Export Style



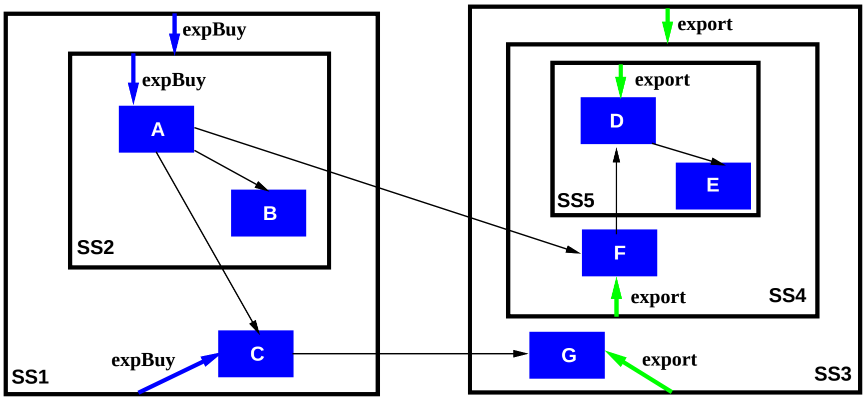
Examples of More Styles



Style 1) Unconstrained Access of Clients and Suppliers



Style 2) Constrained Access of Suppliers

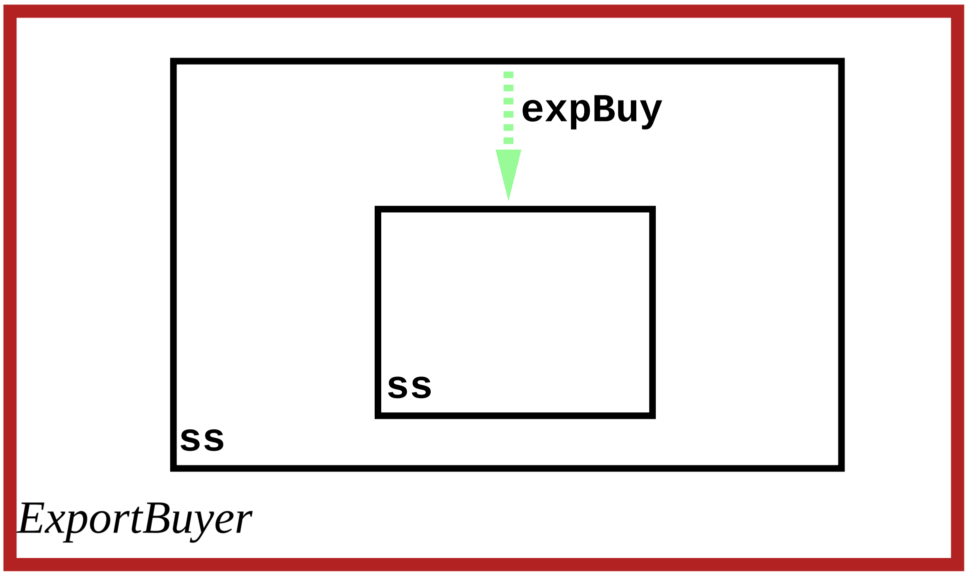


Style 3) Constrained Access of both Clients and Suppliers

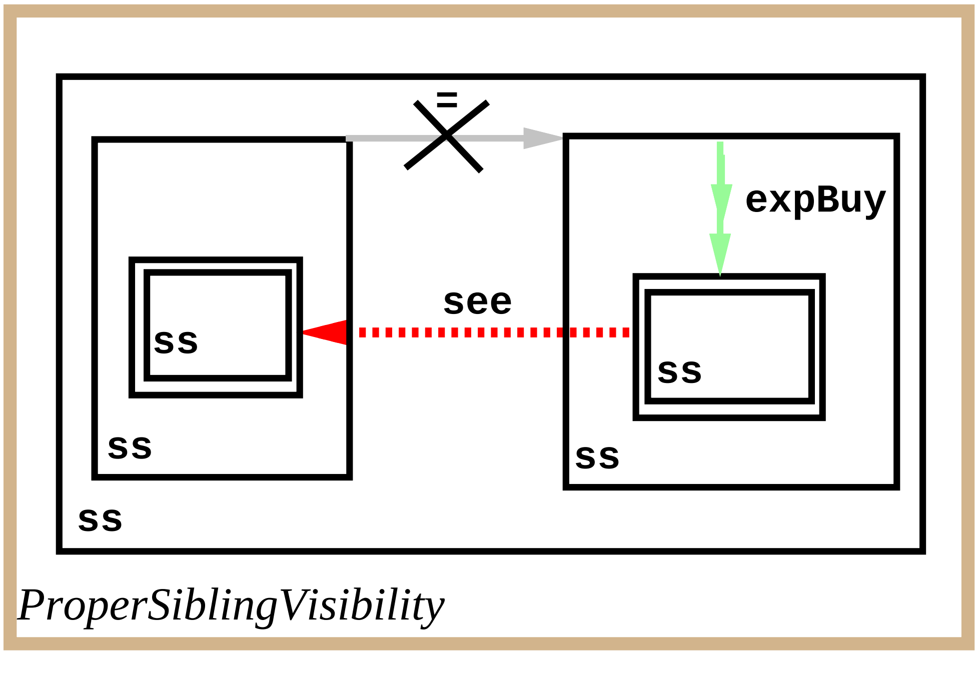
[Mancoridis, 1996b]

Export Buyer Style

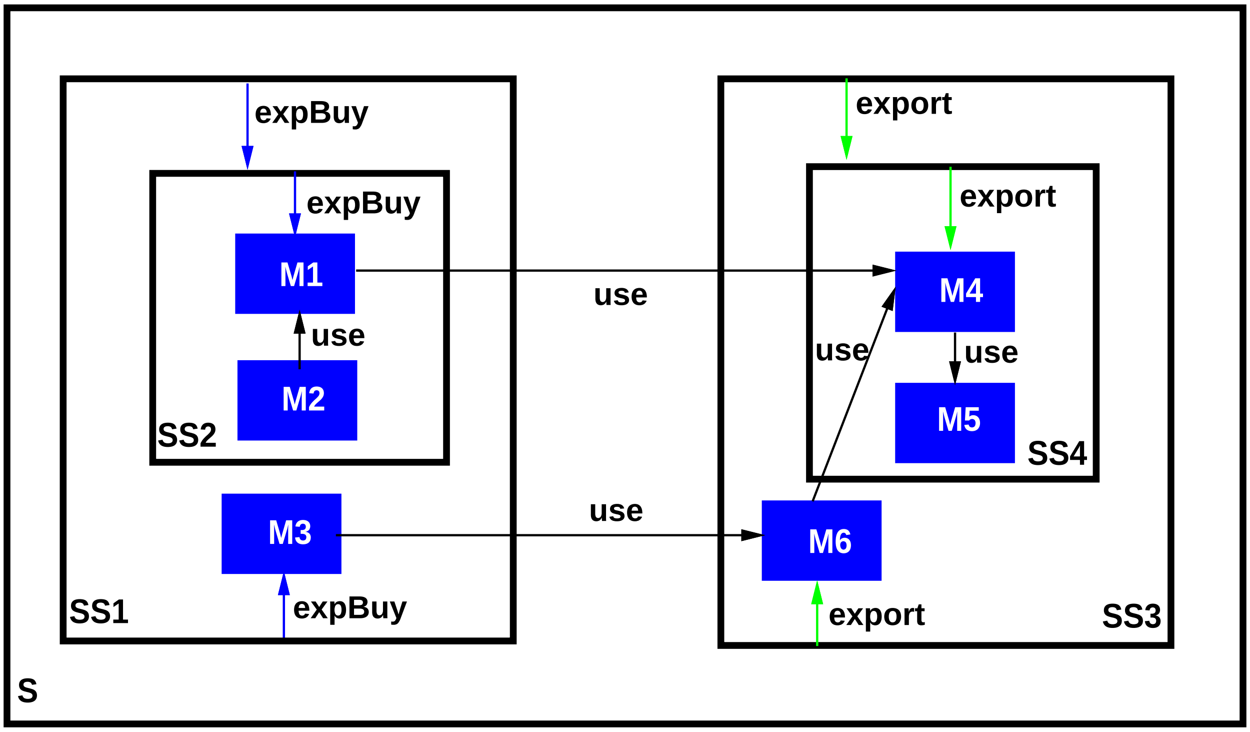
PERMIT



DEFINE

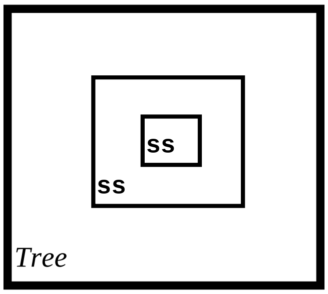


Export/Export Buyer Example

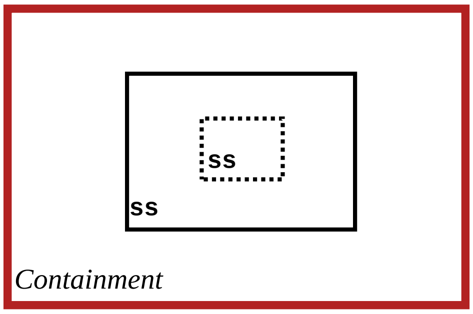


Export/Export Buyer Style

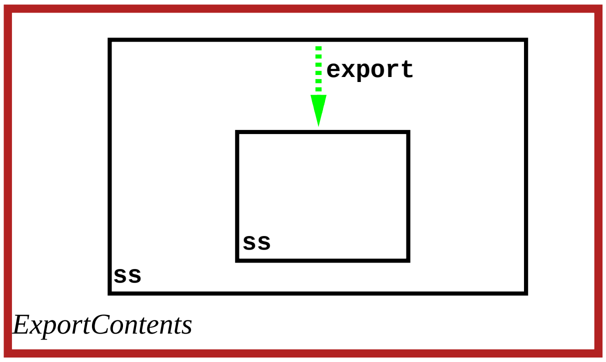
CONSTRAIN



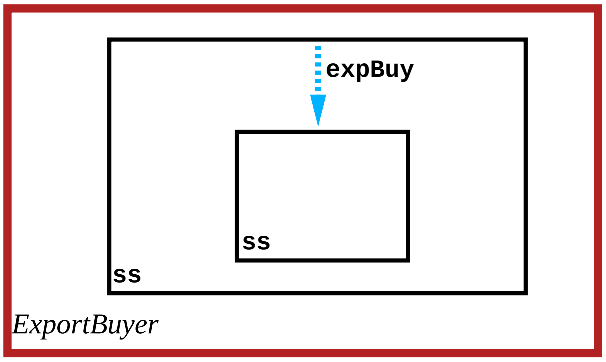
PERMIT



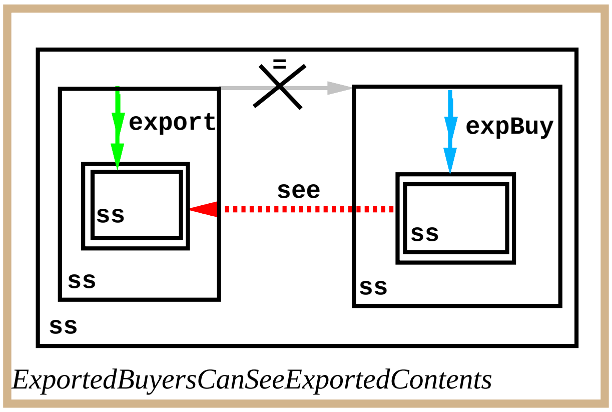
PERMIT



PERMIT



DEFINE



Behavioral Specification Topics

- State Transition Diagrams
- Higraphs and Statecharts
- Petri Nets

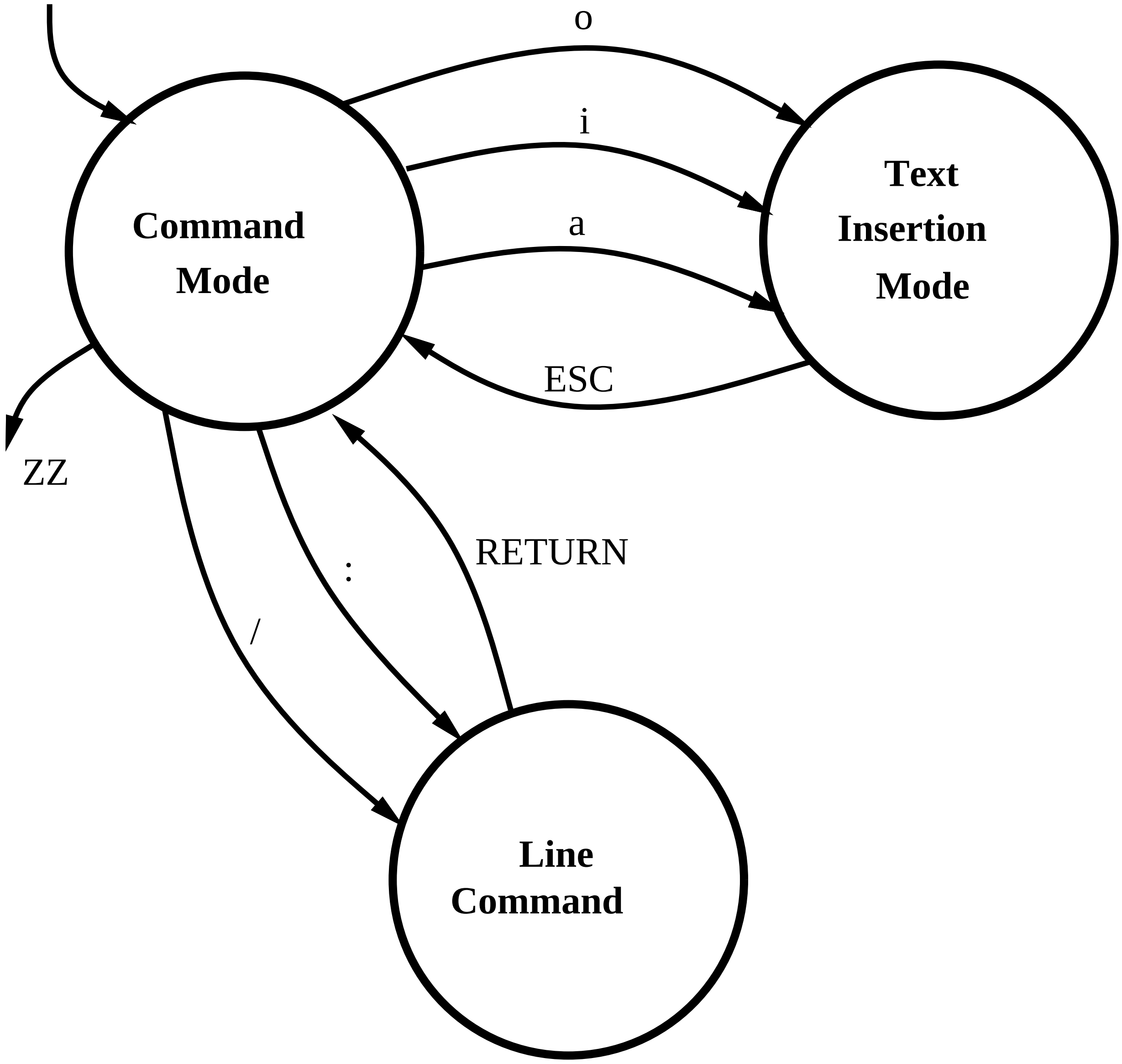
State Transition Diagrams

State Transition Diagrams (STD)

- Systems can be considered as existing in a finite set of possible states, with external events being the triggers that can lead to transitions between the states.
- Since most systems have many states, a partial model of the system may be a good compromise.
- State transition diagrams are the cornerstone of more powerful diagrams for specifying system behavior, such as Petri Nets and State Charts.

Material based on [Budgen, 1994].

Partial Unix vi STD



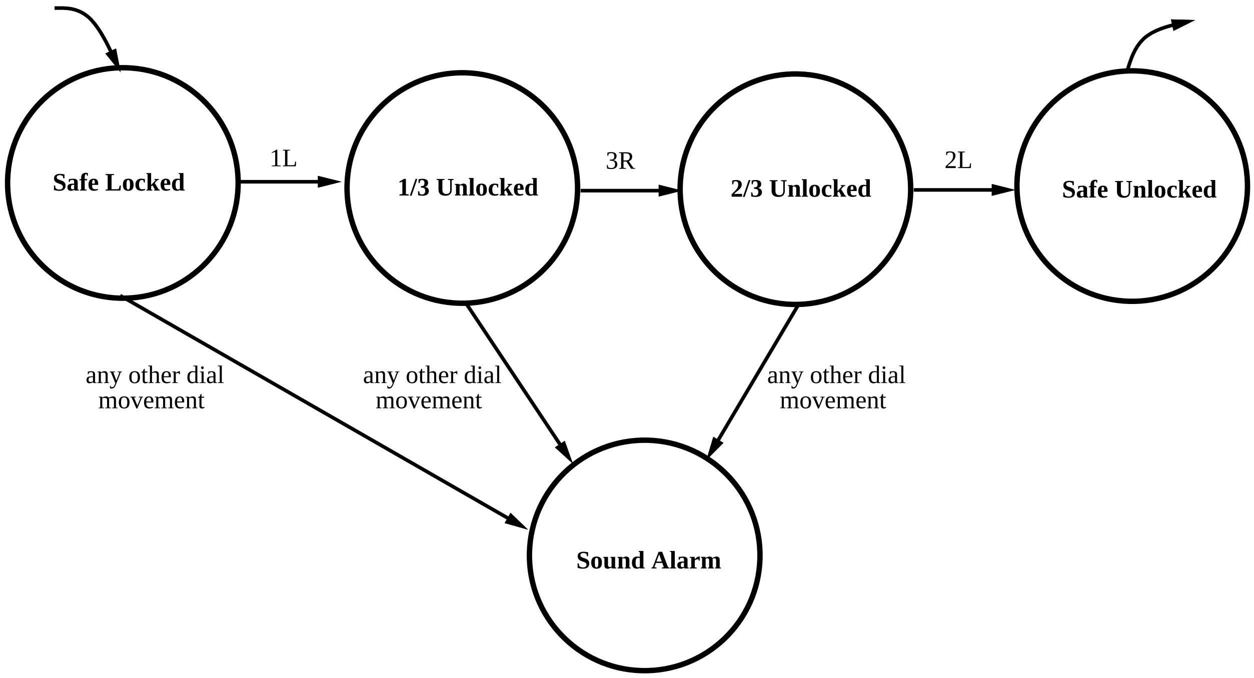
Formal Definition of an STD

$$\mathrm{STD} = (\mathrm{Q}, \Sigma, \mathrm{q}_0, \delta, \mathrm{F})$$

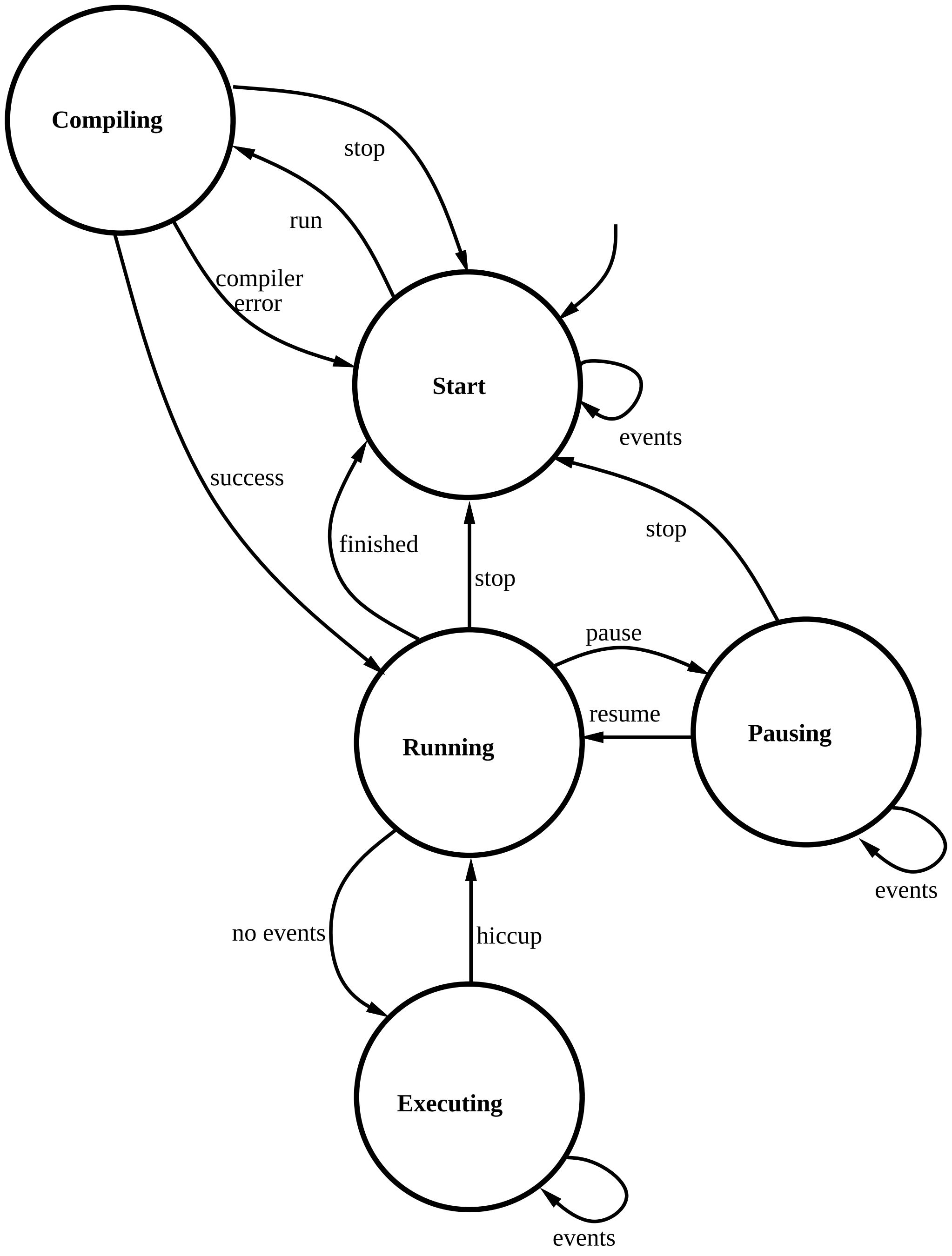
Where:

- Q is a set of states
- Σ is an input alphabet
- $\mathrm{q}_0 \in \mathrm{Q}$ is the start state
- δ is a transition function $\delta : \mathrm{Q} \times \Sigma \rightarrow \mathrm{Q}$
- $\mathrm{F} \subseteq \mathrm{Q}$ is the set of final states

Combination Safe STD



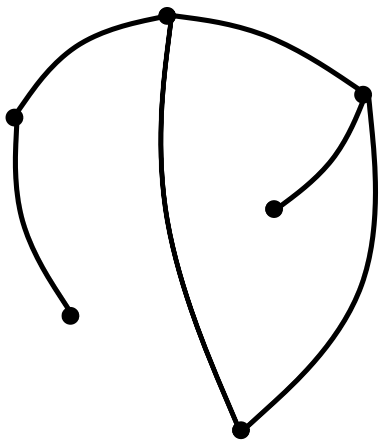
Programming Environment STD



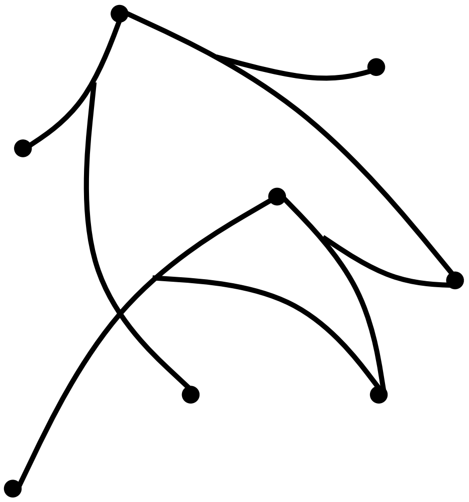
Higraphs and Statecharts

Higraphs

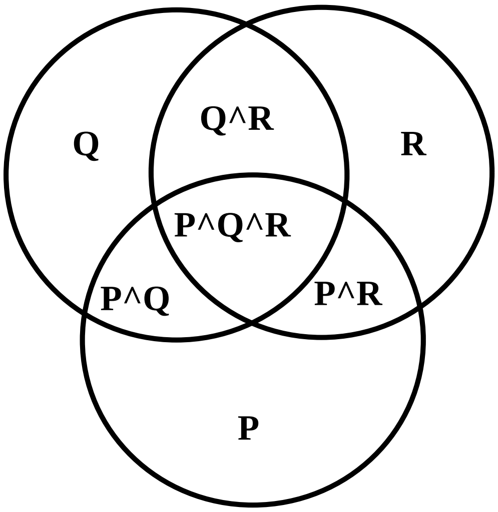
Higraphs [Harel, 1988] are based on Euler graphs, hypergraphs, and Venn diagrams.



Graph



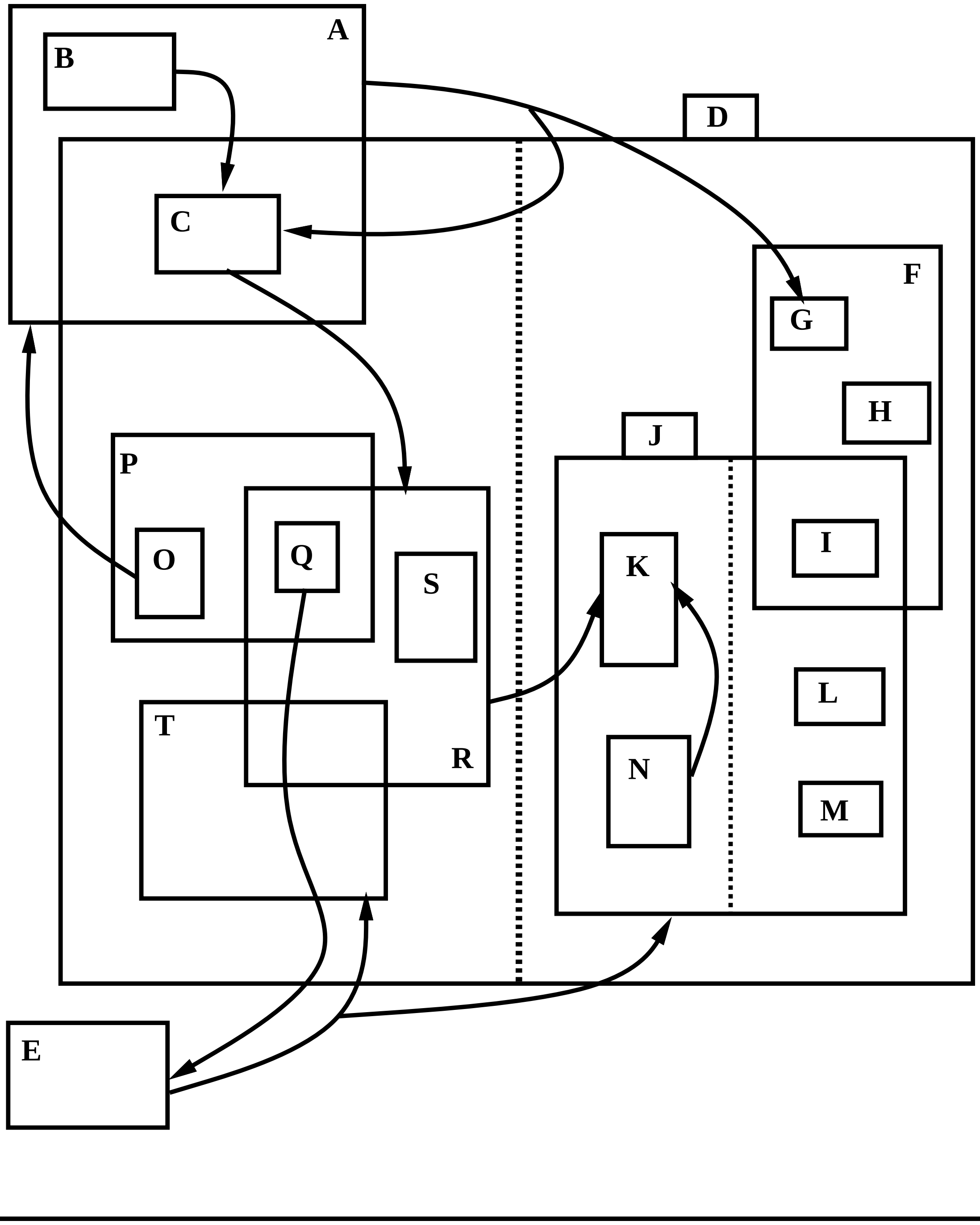
Hypergraph



Venn Diagram

Example of a Higraph

Higraphs also support Cartesian products.



Formal Definition of a Higraph

$$H = (B, \sigma, \pi, E)$$

Where:

- B is a finite set of elements (blobs)
- σ is the sub-blob function $\sigma : B \rightarrow 2^B$
- π is the partitioning function $\pi : B \rightarrow 2^{B \times B}$ $(2^{B \times B} = 2^B \times \ldots \times 2^B)$
- E is the set of edges $E \subseteq B \times B$

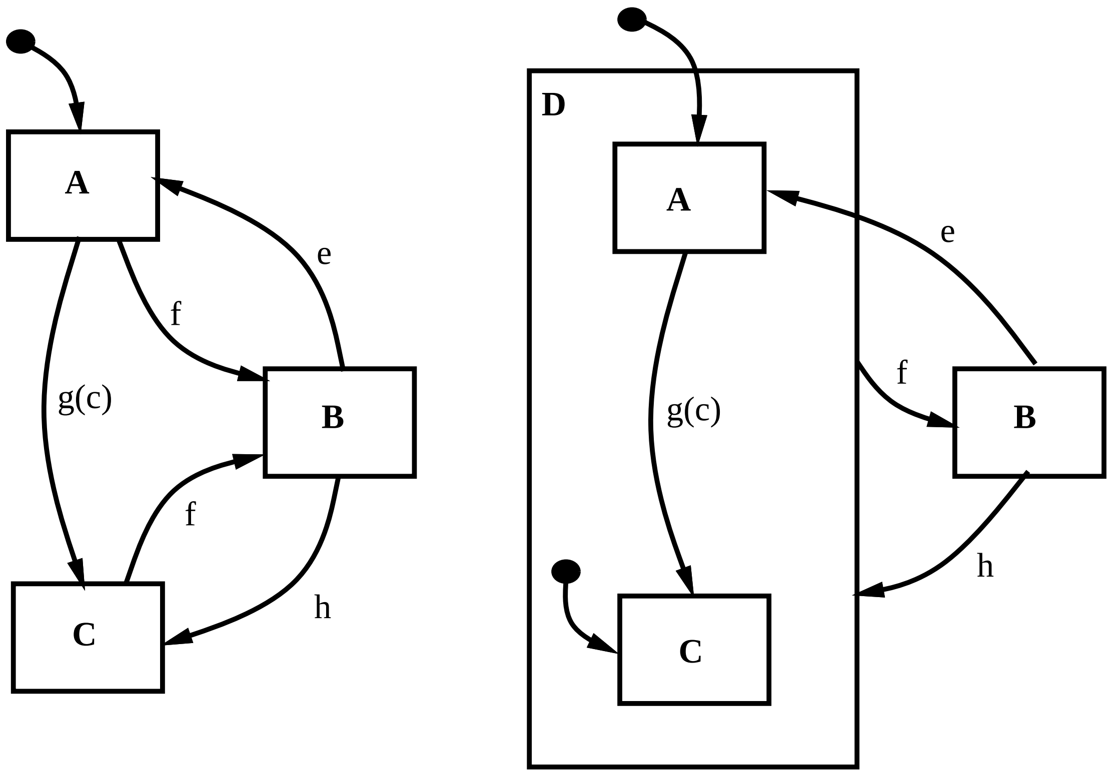
Specialized Higraphs: State Charts

State Charts are a higraph-based extension of standard state-transition diagrams, where blobs represent states and arrows represent transitions.

State Charts =

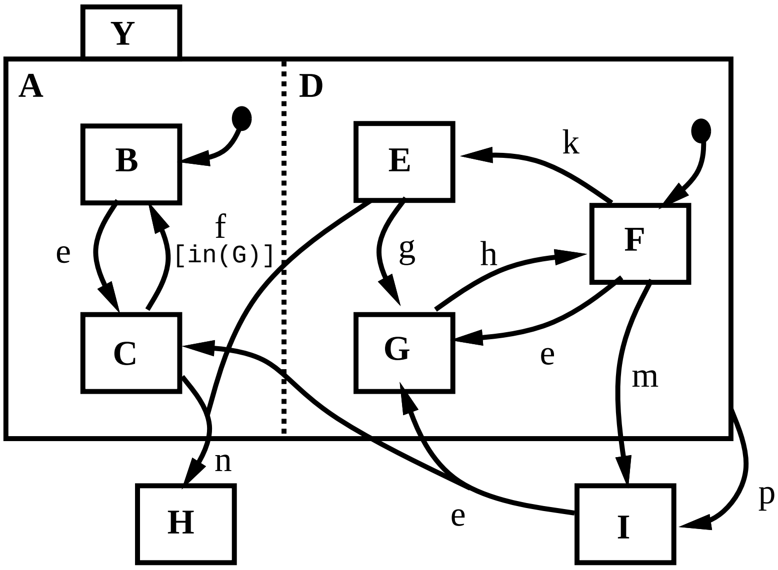
state diagrams +
depth +
orthogonality +
broadcast communication

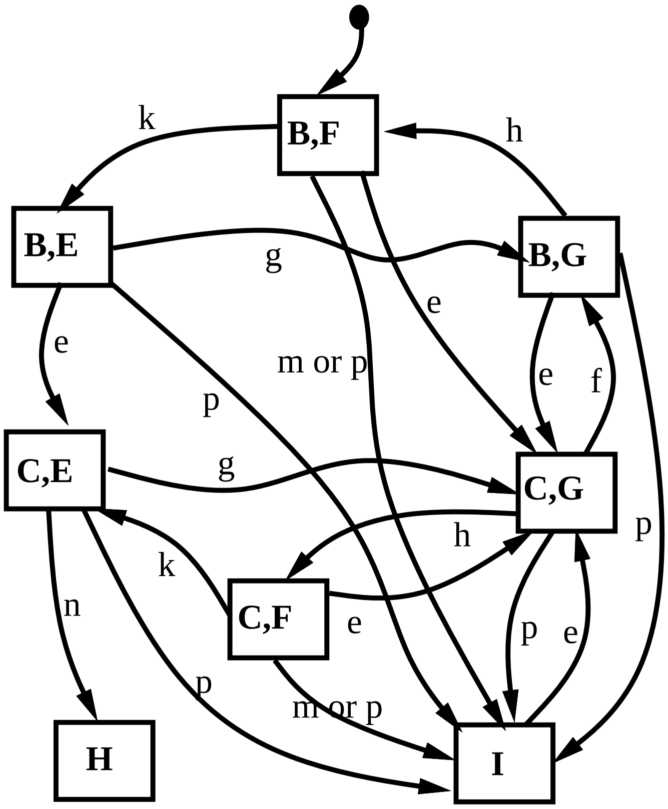
Depth of State Charts



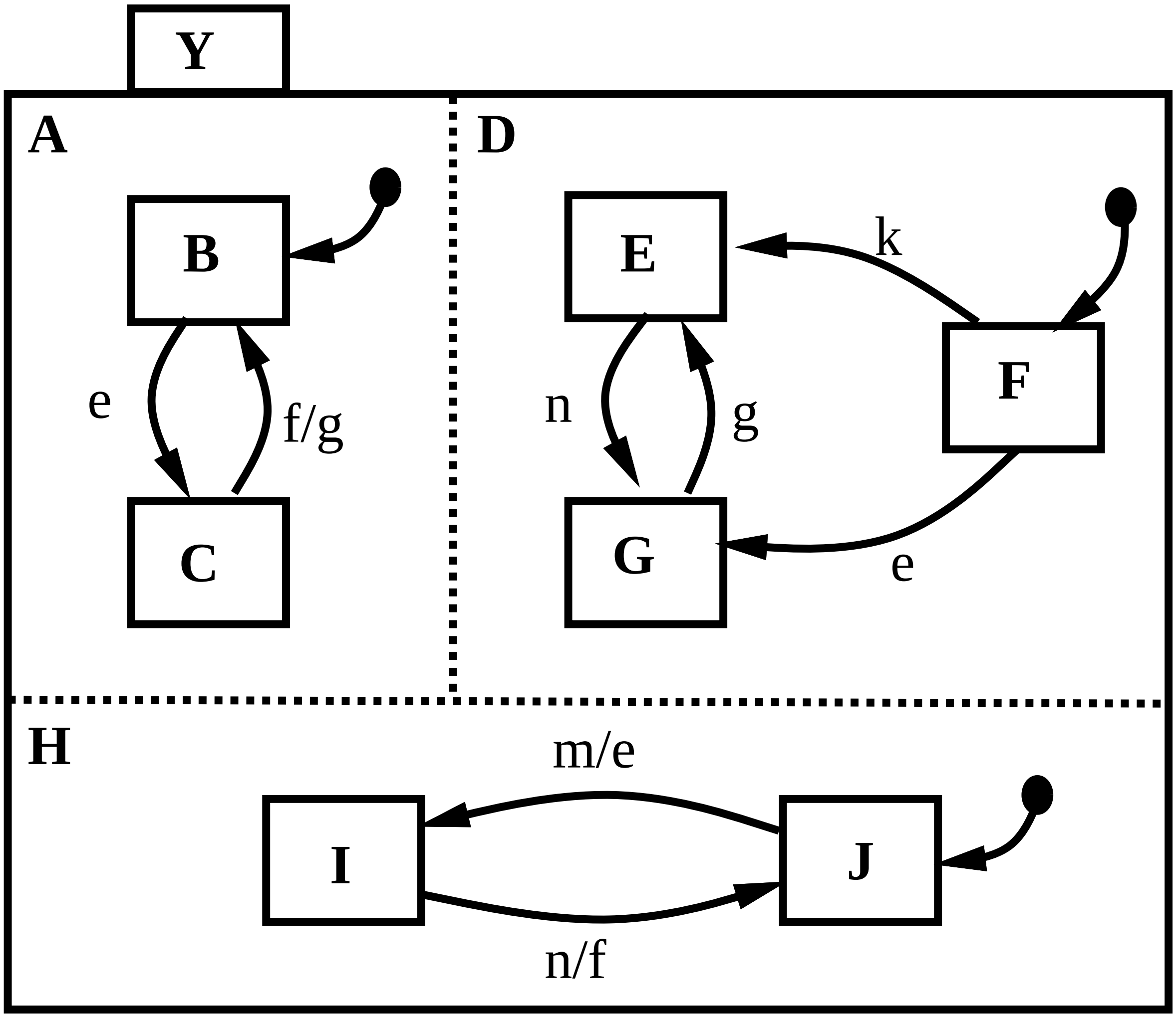
- e, f, g, h: are events that trigger the transitions
- g(c): is transition by event g when condition c is true
- Being in state D means being in one of states A or C
- The f arrow leaving D applies to both A and C
- A is the default state
- C is the default state when in D

Orthogonality of State Charts

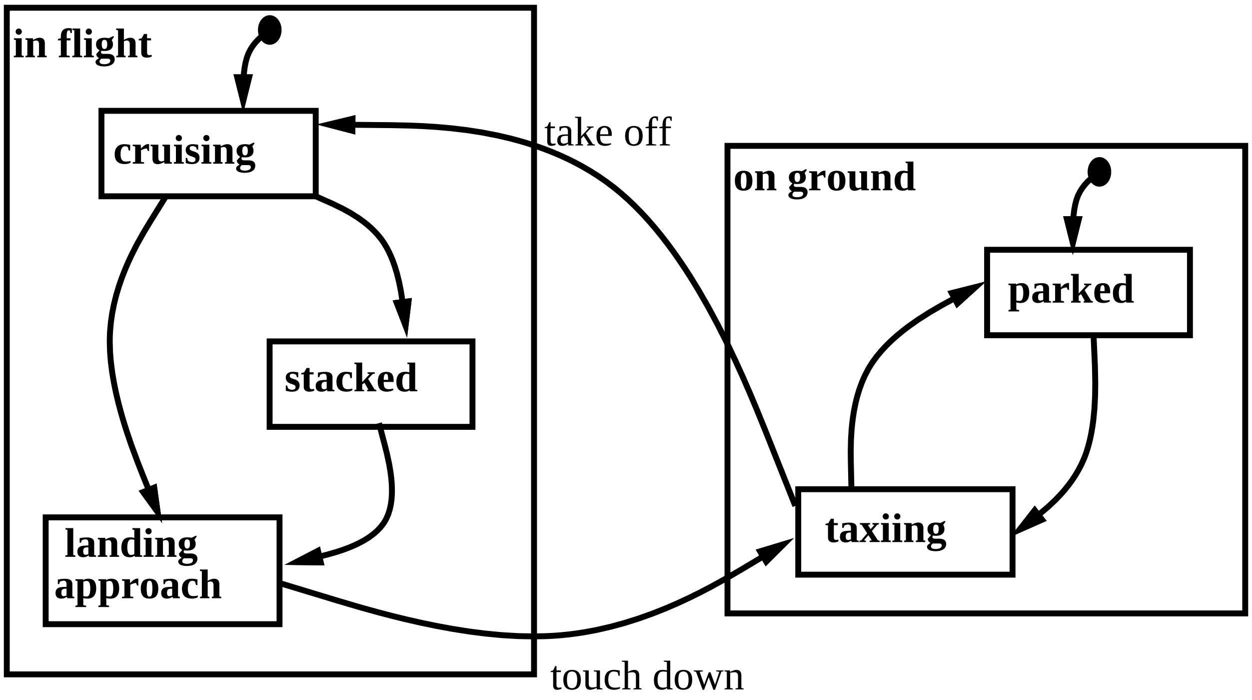




Broadcasting of State Charts

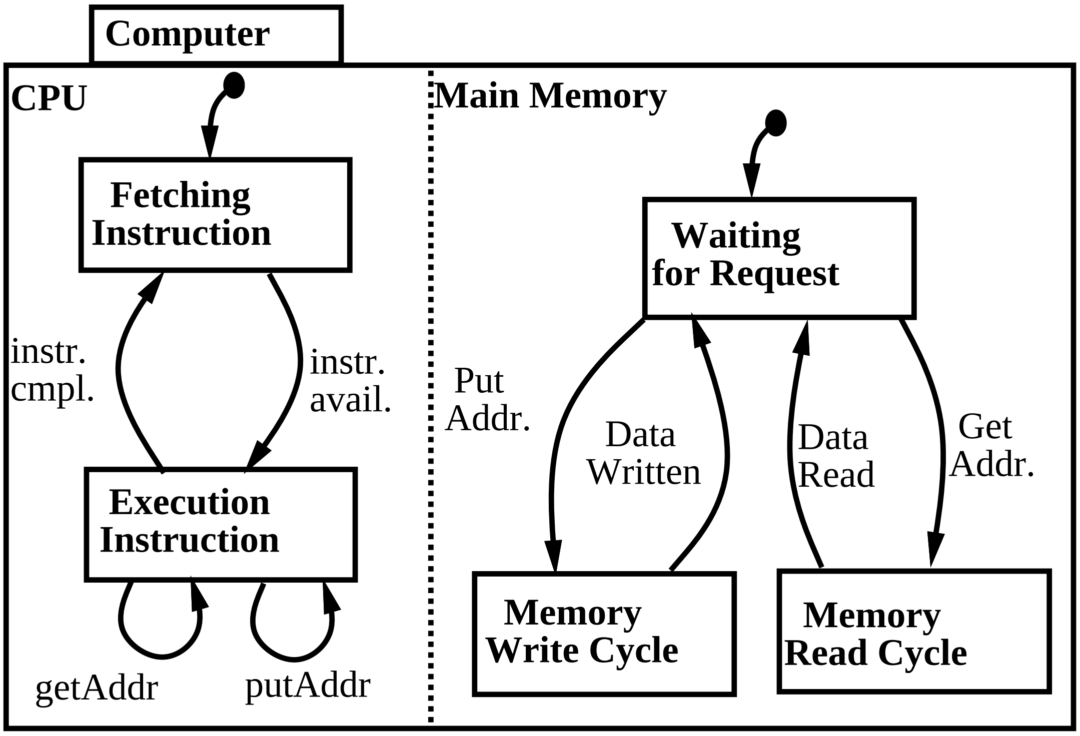


State Chart Describing ATC



[Budgen, 1994]

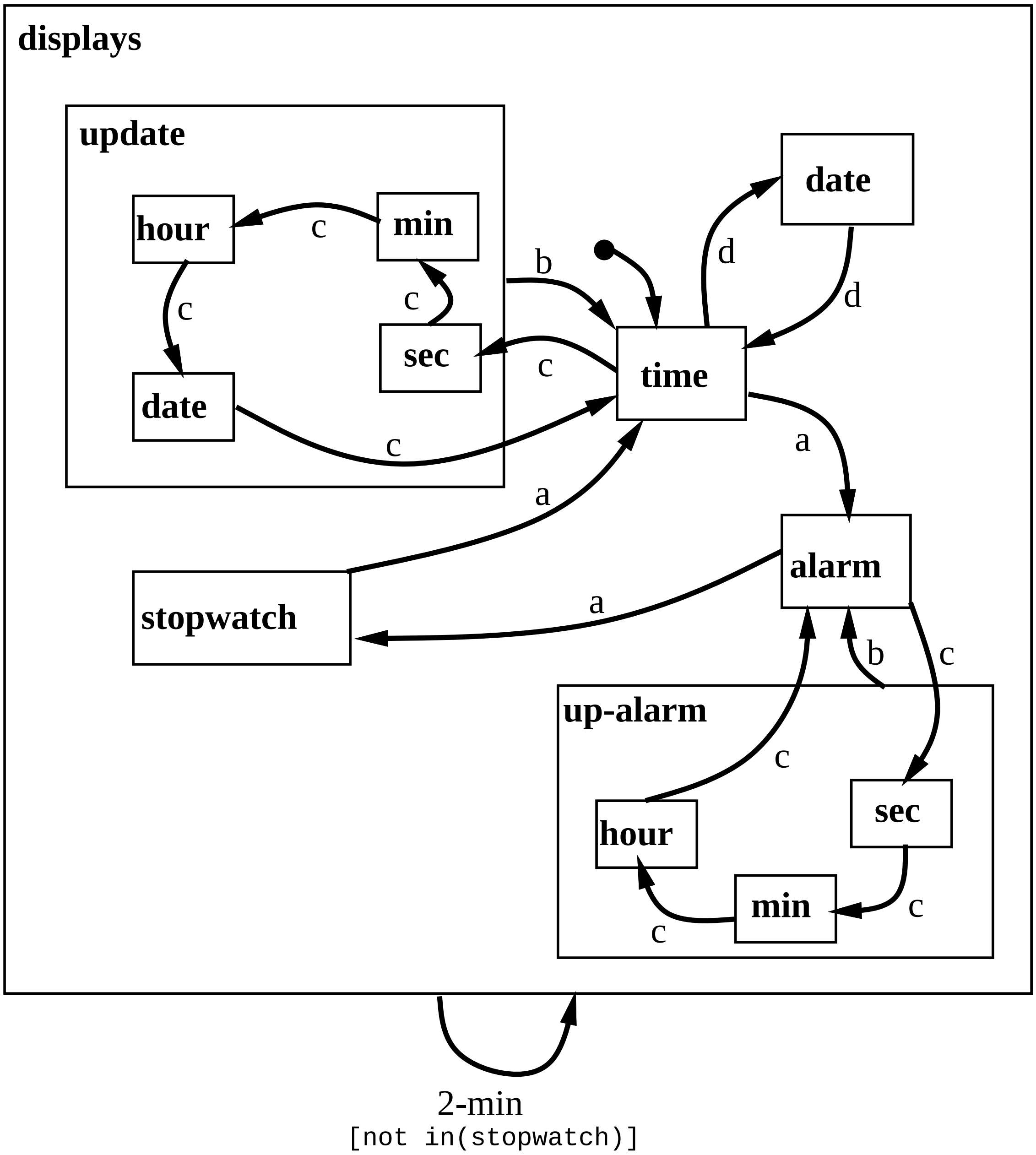
State Chart Describing a Computer



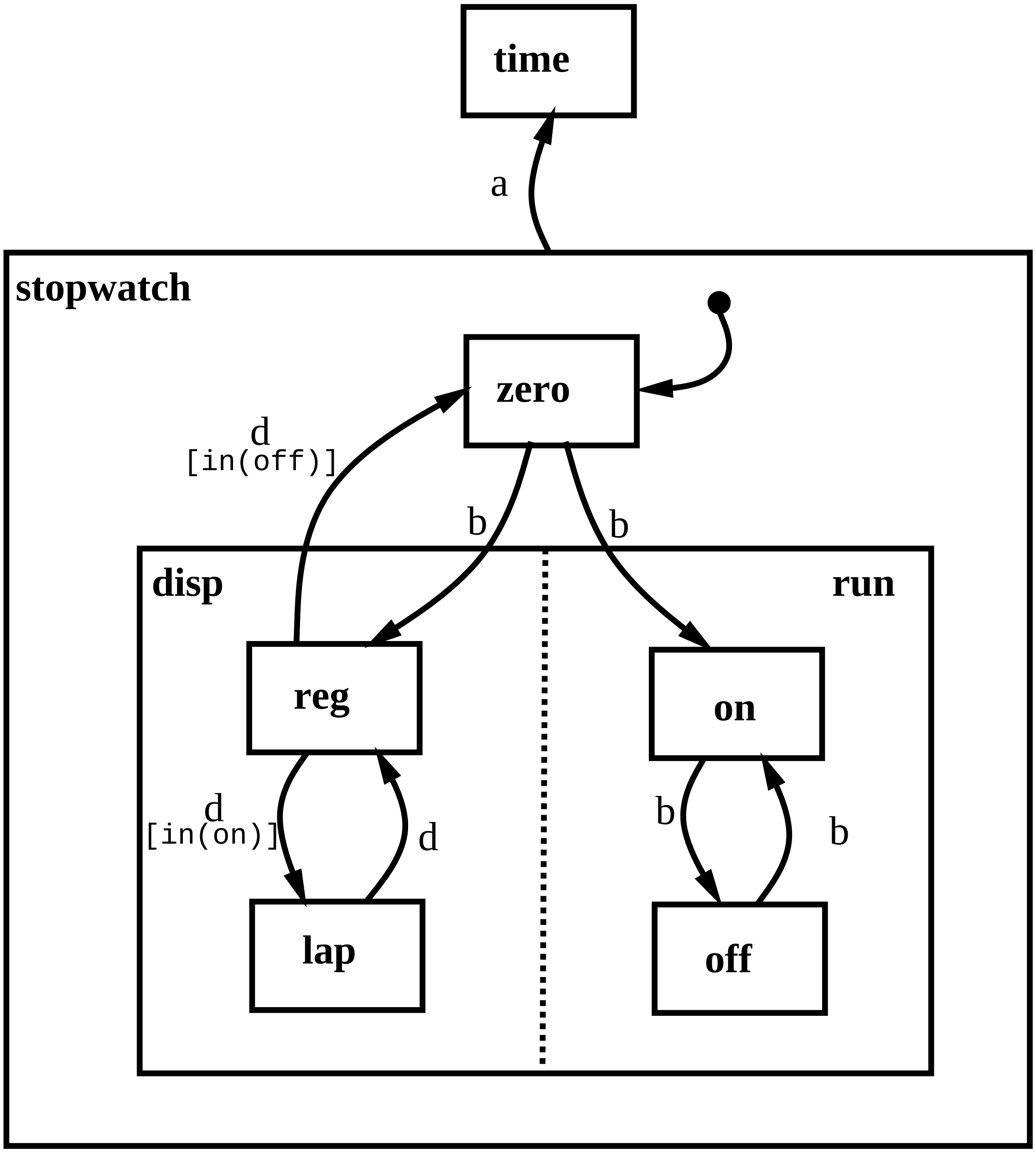
- Main memory may be accessed by CPU or device drivers

[Budgen, 1994]

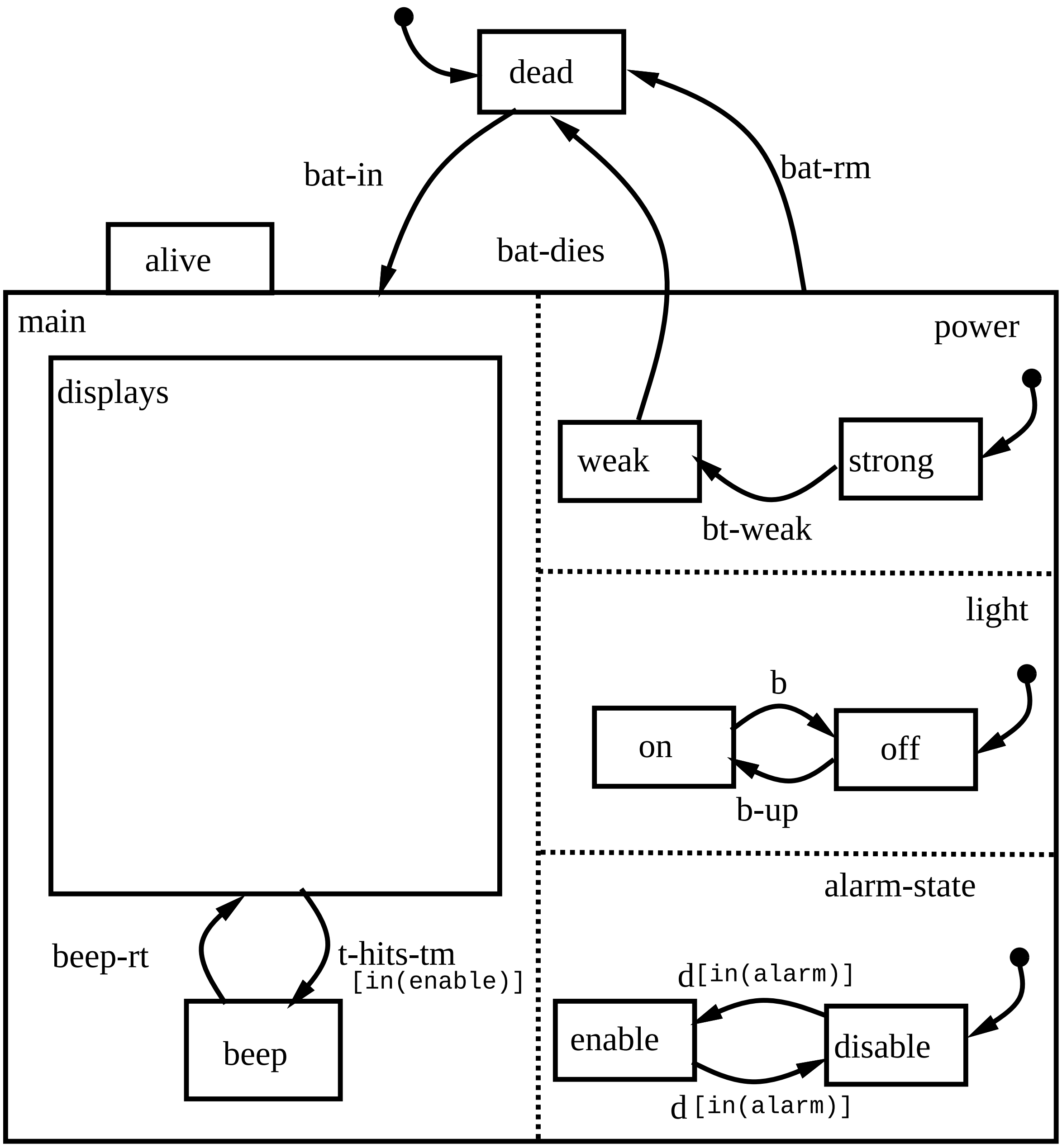
Display State of Digital Watch



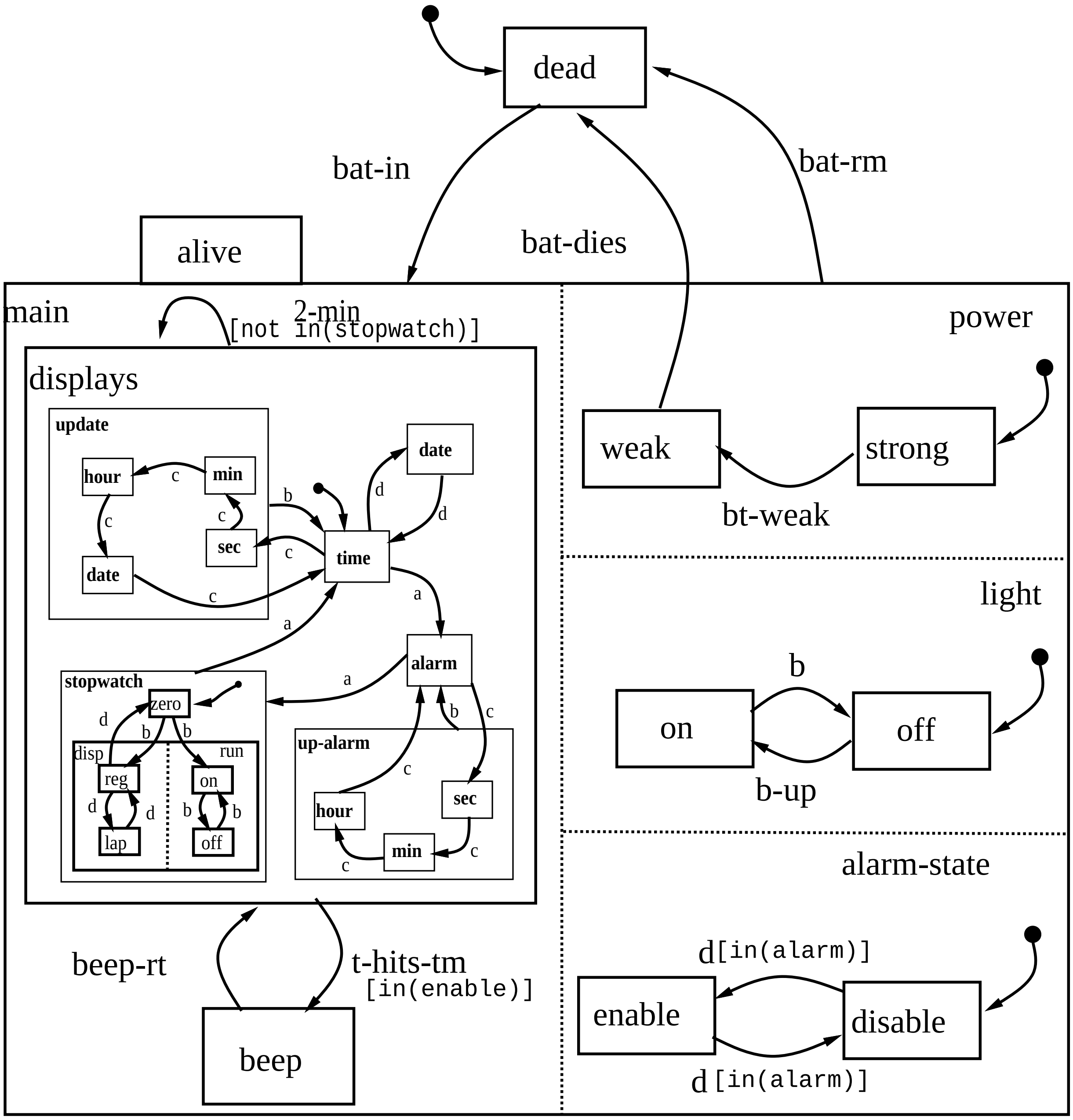
Stopwatch State of Digital Watch



High-Level Desc. of Digital Watch



State Chart of Digital Watch



Petri Nets

Petri Nets: Motivation

A major difficulty with specifying real-time systems is coping with timing. This difficulty can manifest itself in many different ways:

- deadlock,
- synchronization problems,
- starvation.

Timing problems can arise as a consequence of:

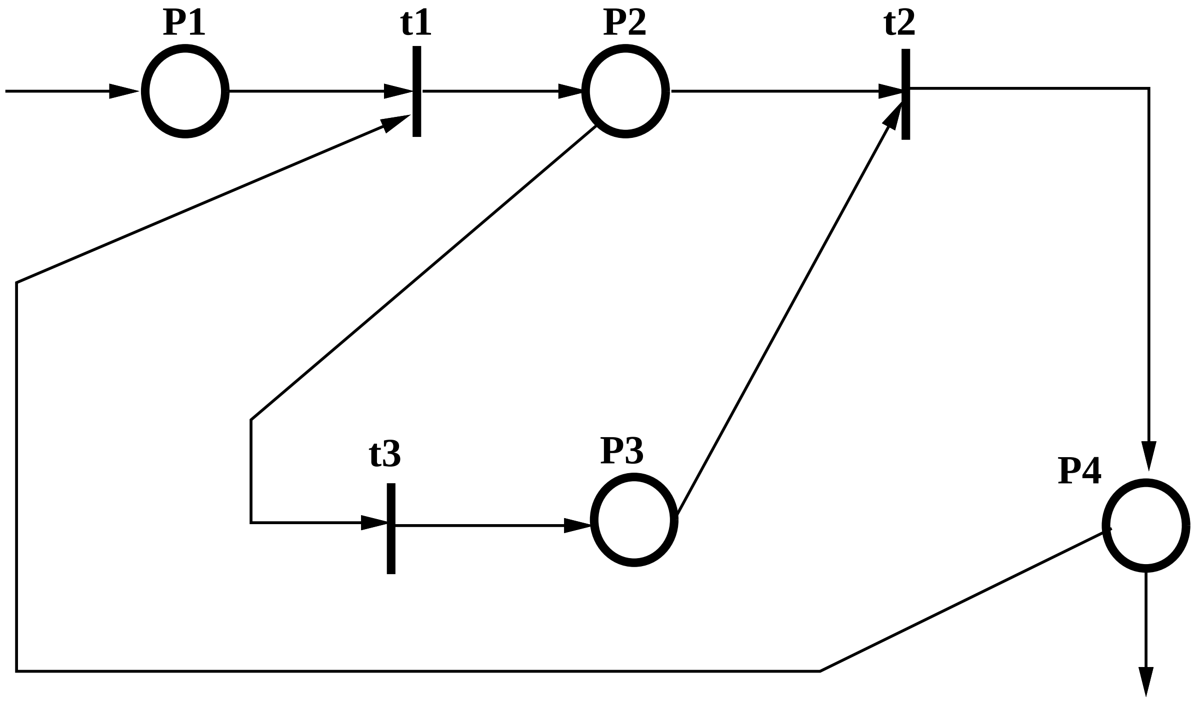
- poor design,
- faulty implementation,
- poor specifications.

Material based on [Rosenblum and Wolf, 1994, Budgen, 1994, Ghezzi et al., 1991].

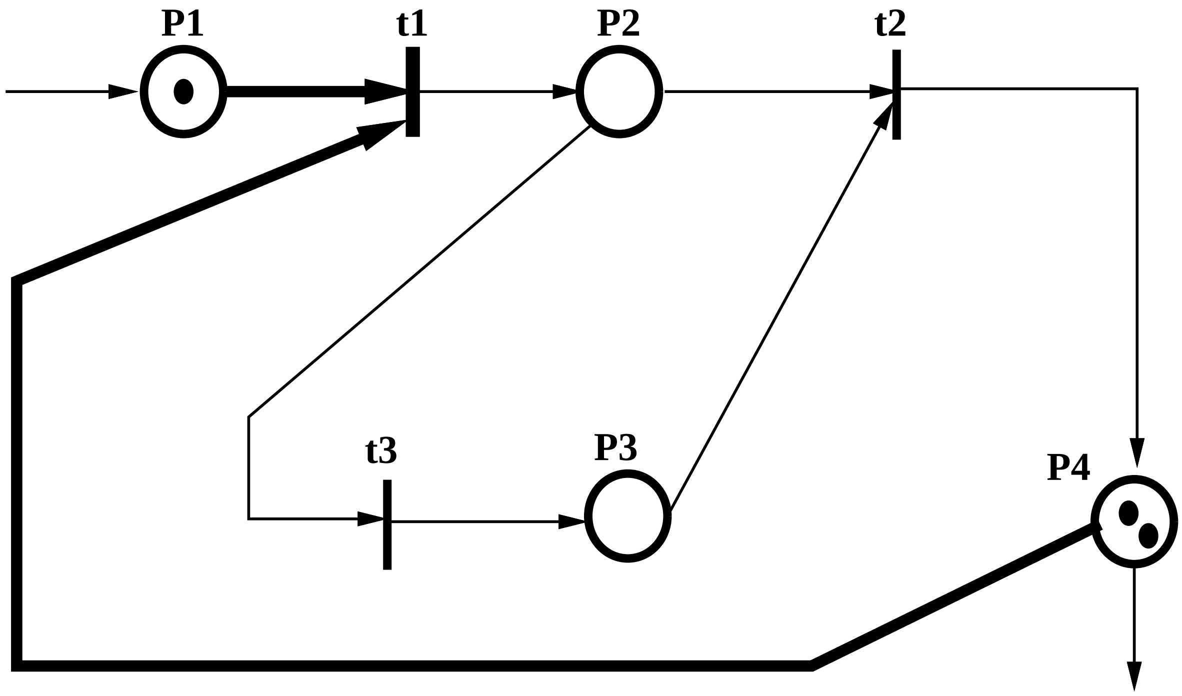
Petri Nets

- capture design aspects that are concerned with behavioral features of a system,
- are primarily concerned with modeling the asynchronous behavior of a number of cooperating processes in a non-deterministic manner,
- are generally used as an auxiliary design notation rather than as a primary form,
- are executable and hence can be analyzed to discover various system properties such as deadlock and unreachable states,
- usually ignore data,
- are, in the general case, intractable to analyze.

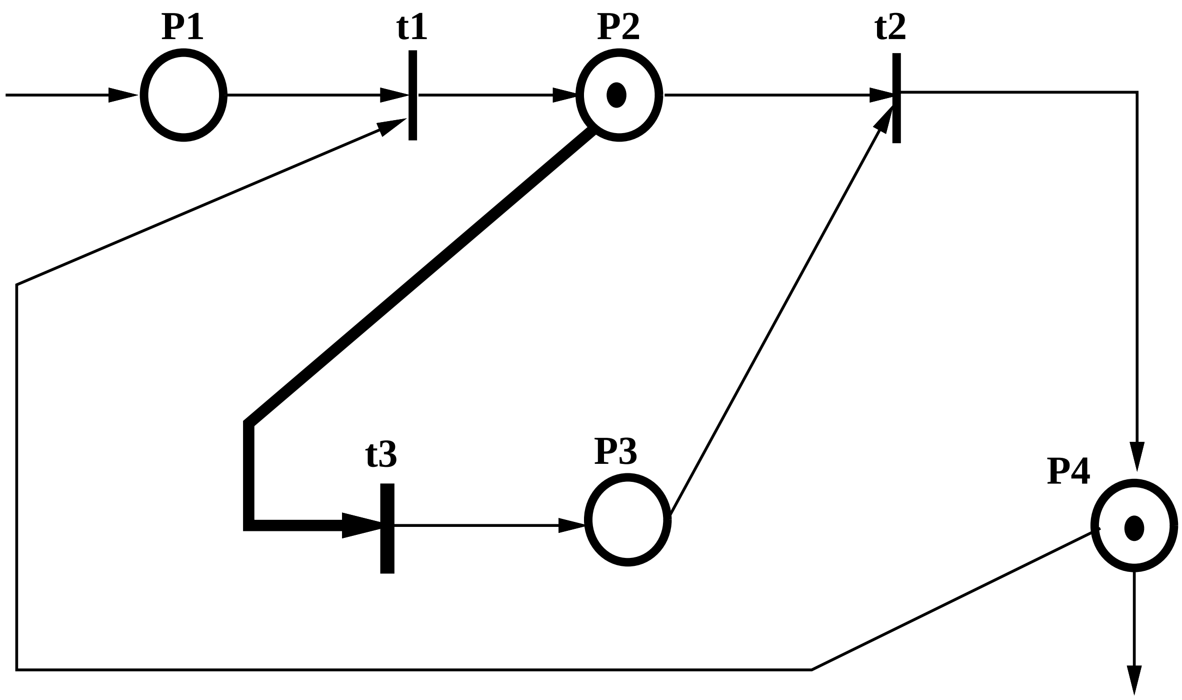
A Simple Petri Net



A Marked Petri Net



A Marked Petri Net After Firing t1



Definition of a Petri Net

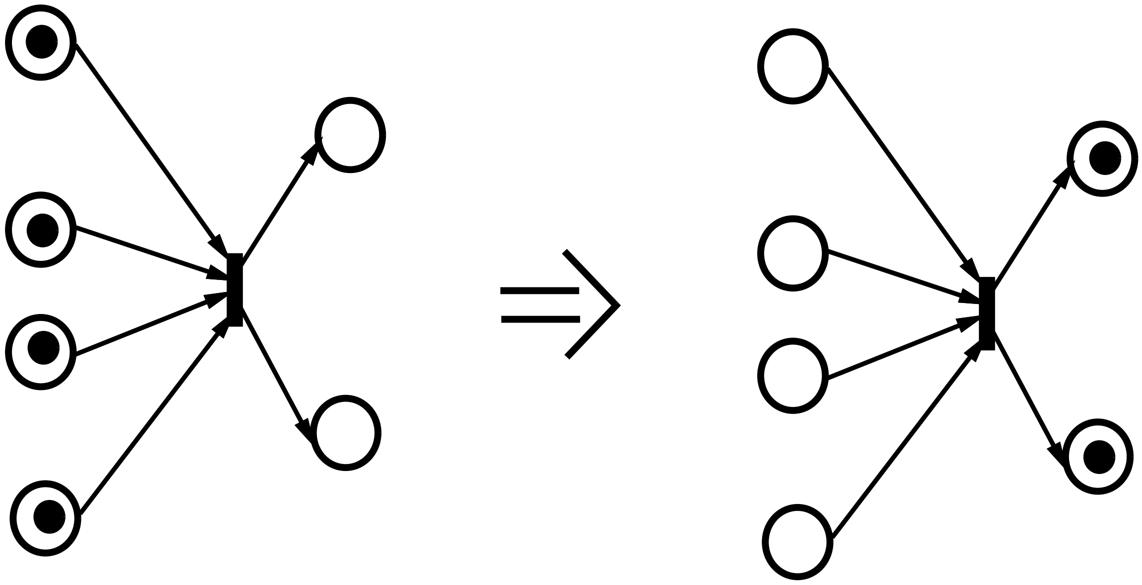
$$\mathbf{PNet} = \{\mathbf{P}, \mathbf{T}, \mathbf{A}, \mathbf{M}_0\}$$

- **P** is a finite set of *places* (labelled circles), where a place holds tokens
- **T** is a finite set of *transitions* (bars), where a transition represents an activity
- **A** is a finite set of directed *arcs*, where an arc connects a place and a transition
- $\mathbf{M}_0$ is the initial *marking* of **N**, where a marking is an arrangement of tokens in places representing state

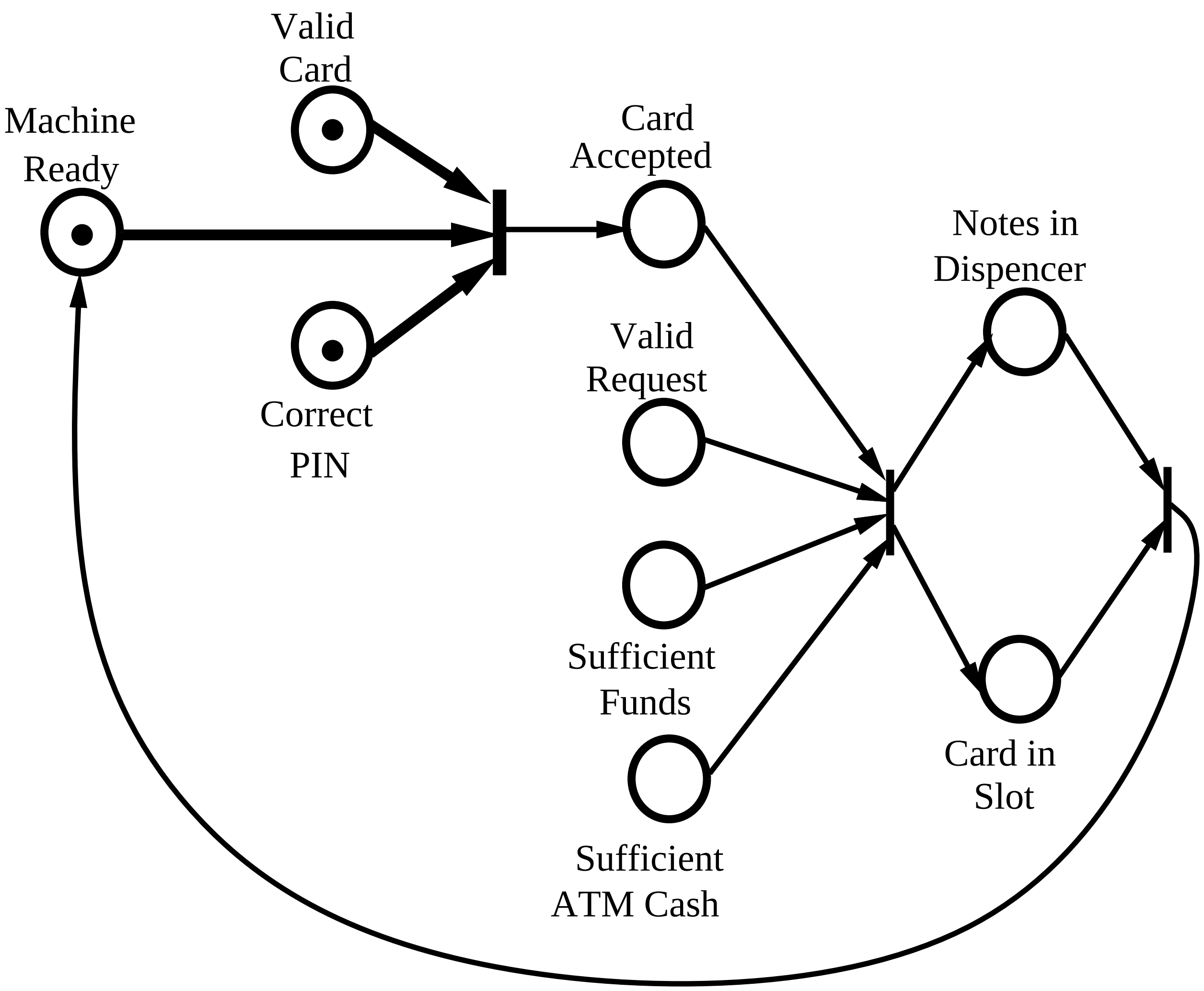
Petri Net Execution Model

- **Input Place:** Place **P** is an *input place* for transition **T** if there is an arc from **P** to **T**
- **Output Place:** Place **P** is an *output place* for transition **T** if there is an arc from **T** to **P**
- **Enabled Transition:** A transition is *enabled* if there is at least one token at each of its input places
- **Firing a Transition:** An enabled transition is non deterministically selected and *fired* by removing one token from each of its input places and depositing one token at each of its output places.
- **Firing Sequence:** A firing sequence $<t_0, t_1, \ldots, t_n>$ such that t_0 is enabled and fired in M_0, t_1 is enabled and fired in M_1, ...

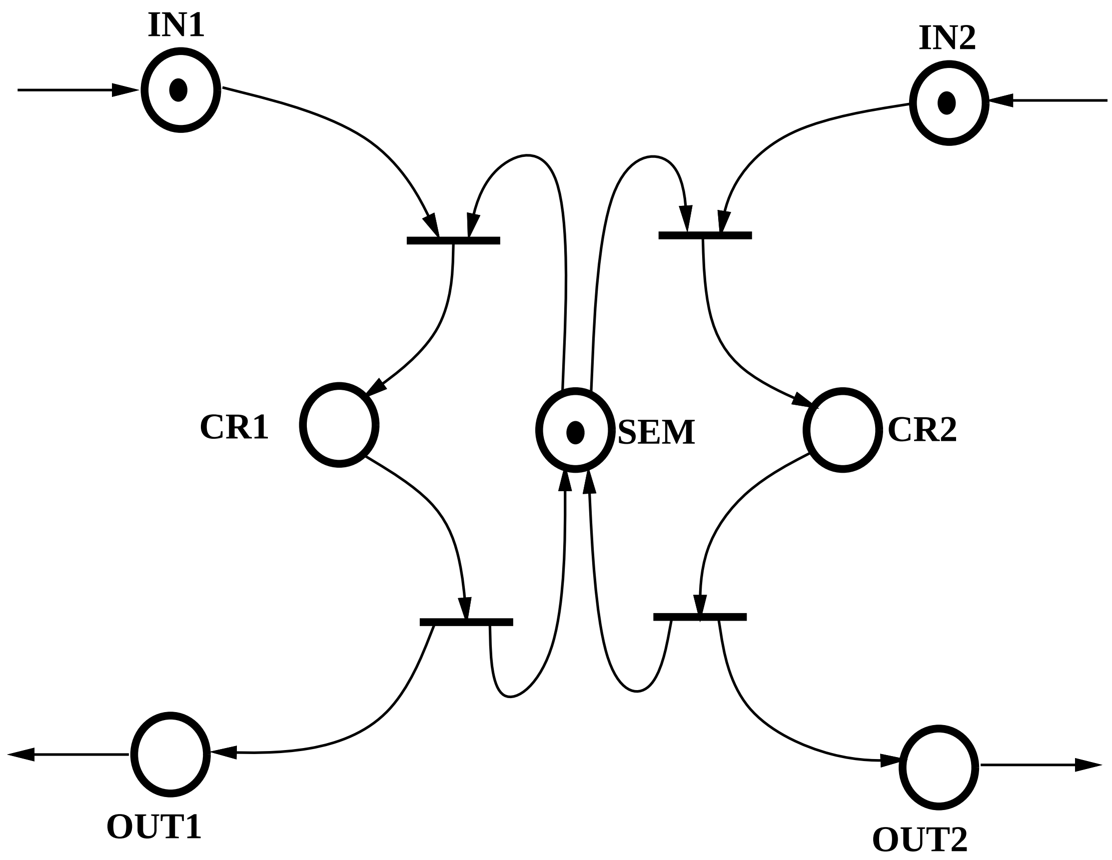
Petri Net Firing



A Petri Net Describing an ATM

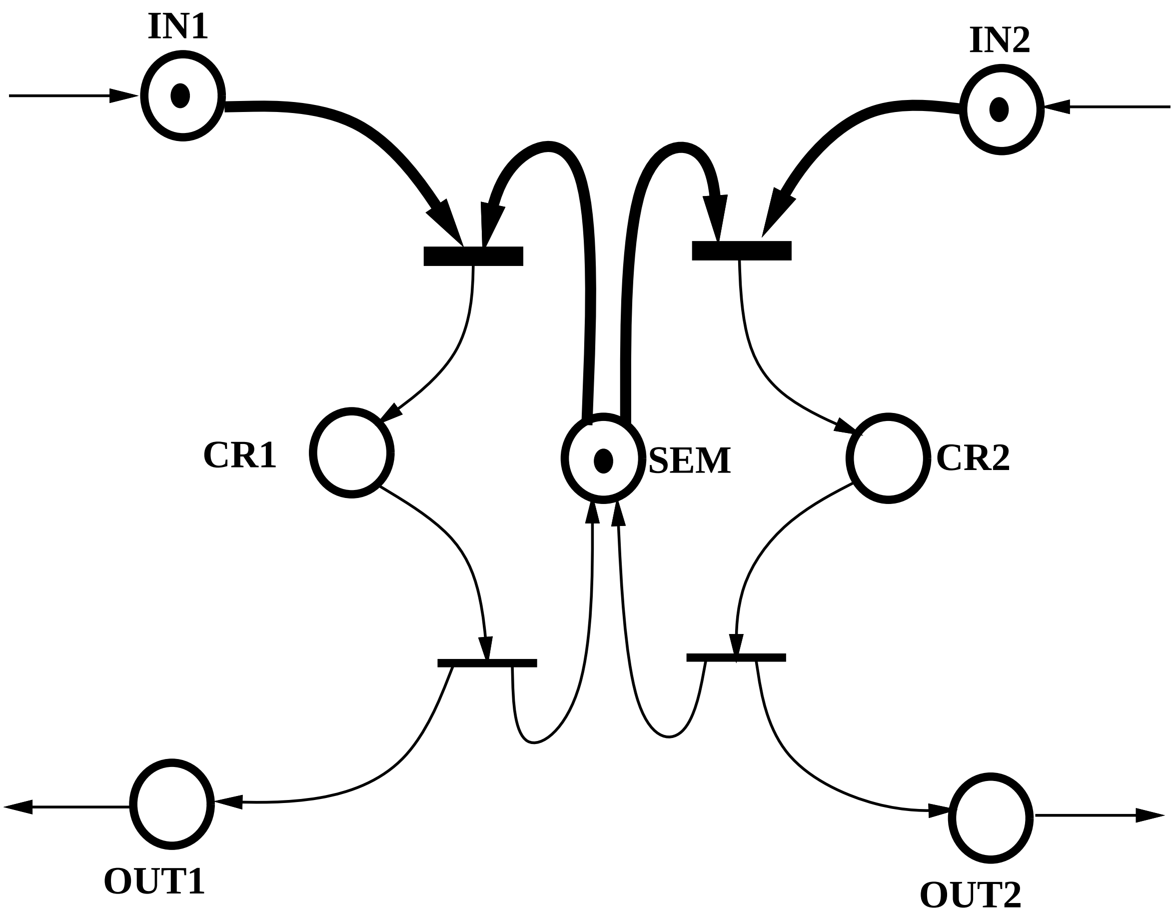


A Marked Petri Net Semaphore

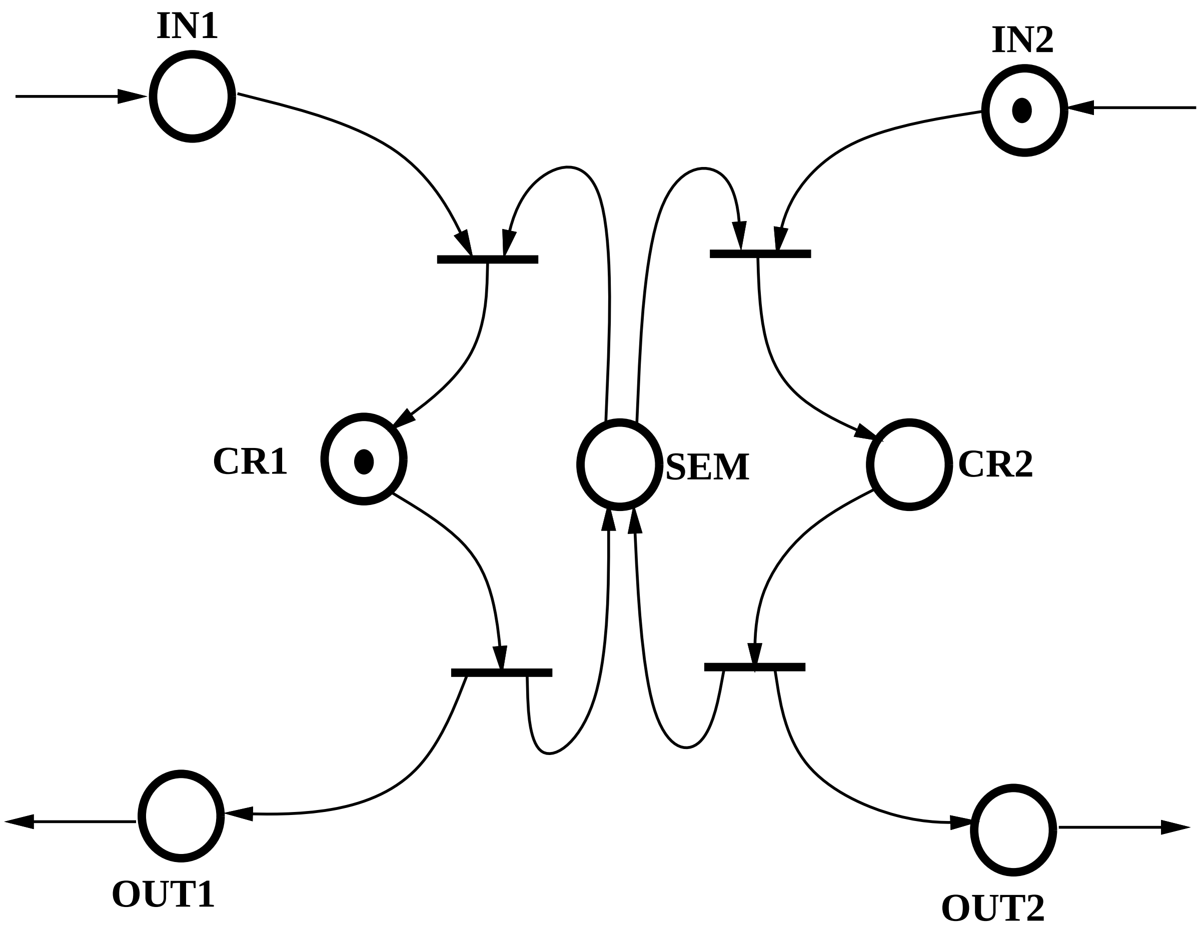


IN = Input of Process
OUT = Output of Process
CR = Critical Region
SEM = Semaphore

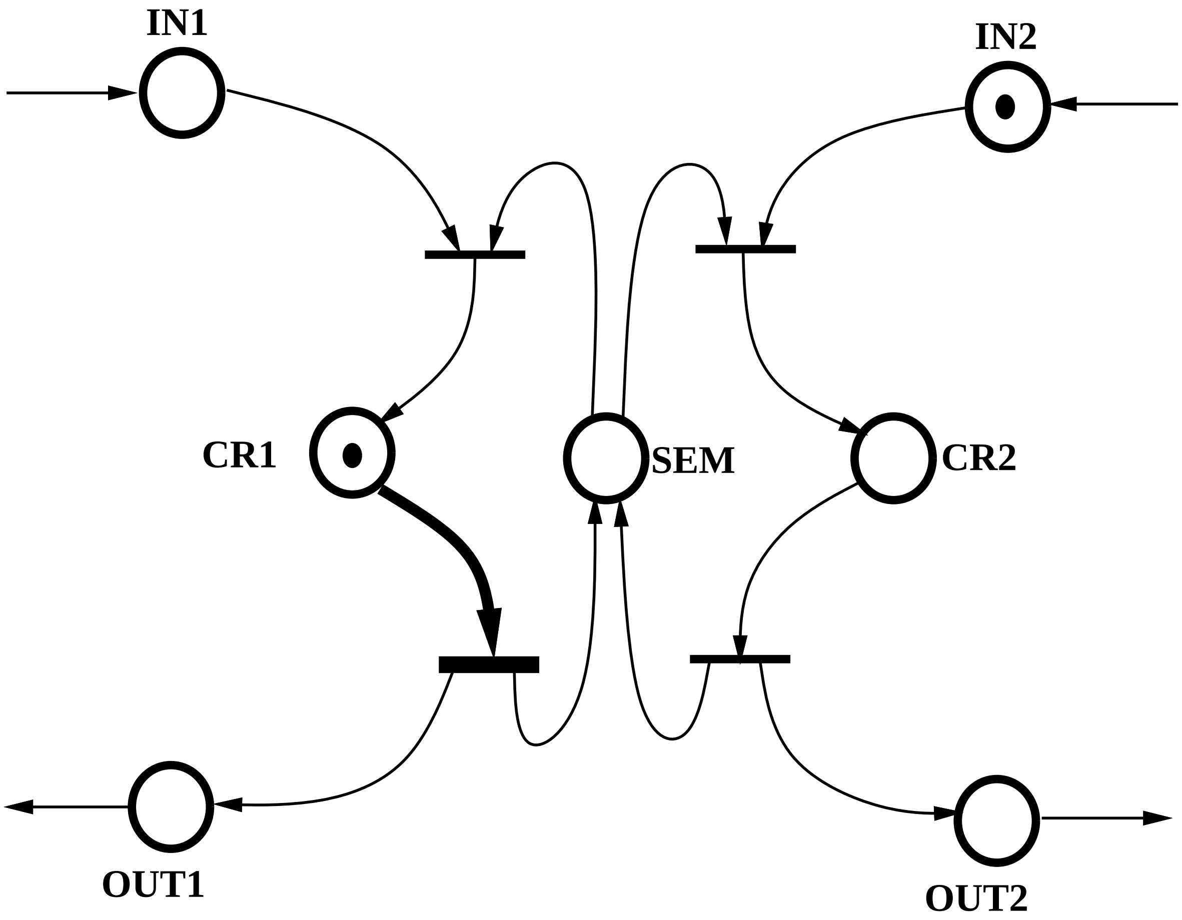
Enabled Transitions



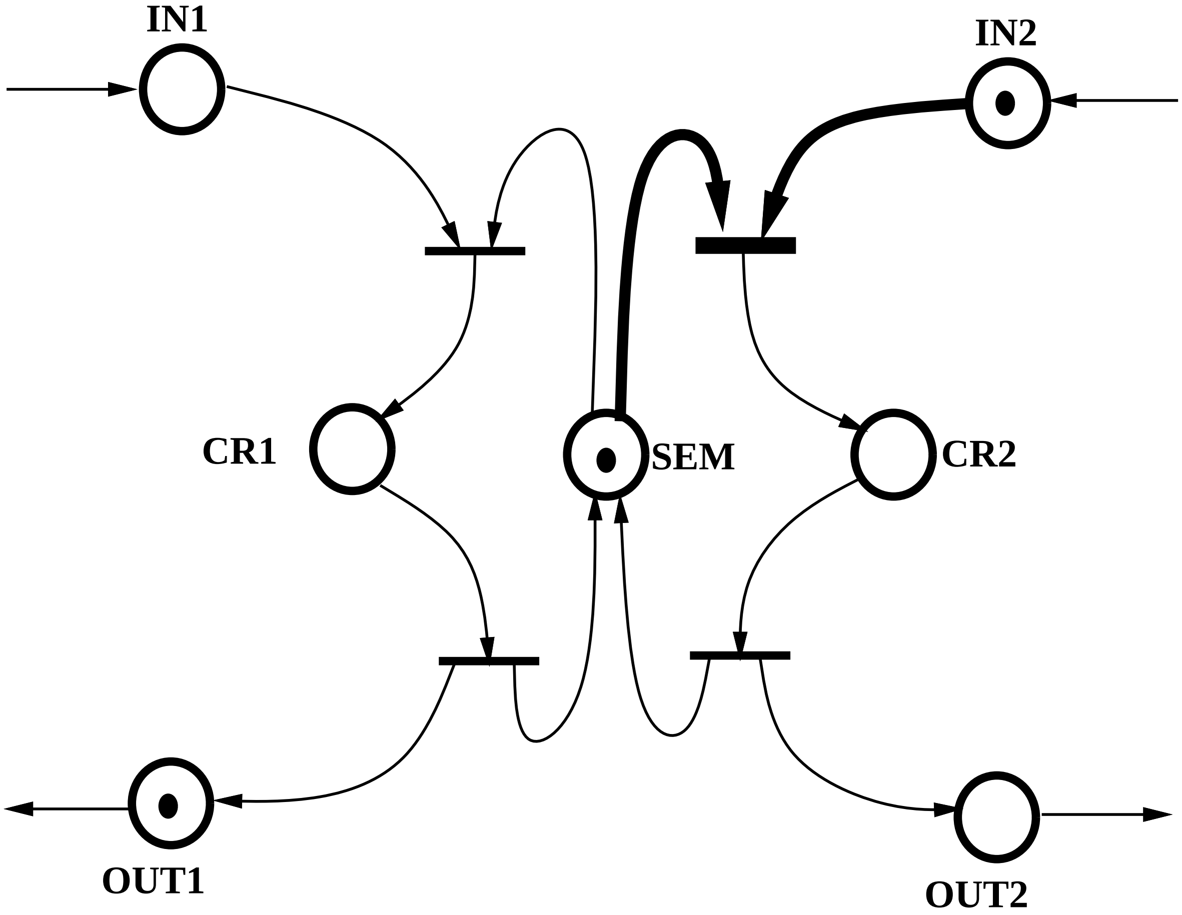
After Non-Deterministic Firing



Enabled Transition



After Firing



Petri Net Static Analysis

Invariants are properties of a Petri net that hold in all markings.

For example, the sum of all tokens in CR1, CR2, and **SEM** are equal to 1 in all reachable markings. That is,

$$|CR1| + |CR2| + |SEM| = 1$$

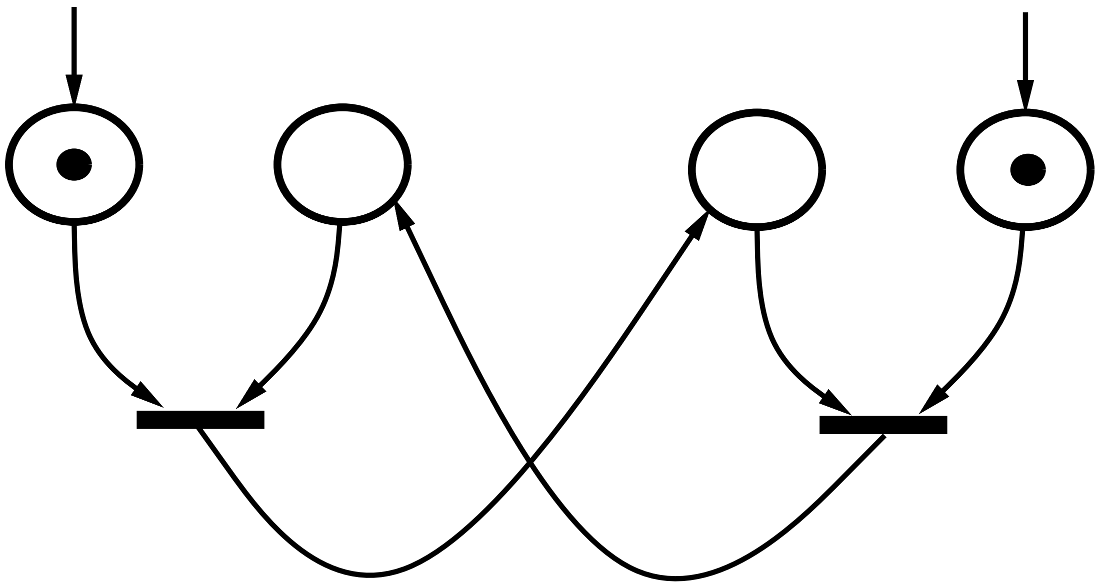
Deadlock and Starvation

A Petri Net with a given marking is said to be in a **deadlock** situation if and only if no transition is enabled in that marking.

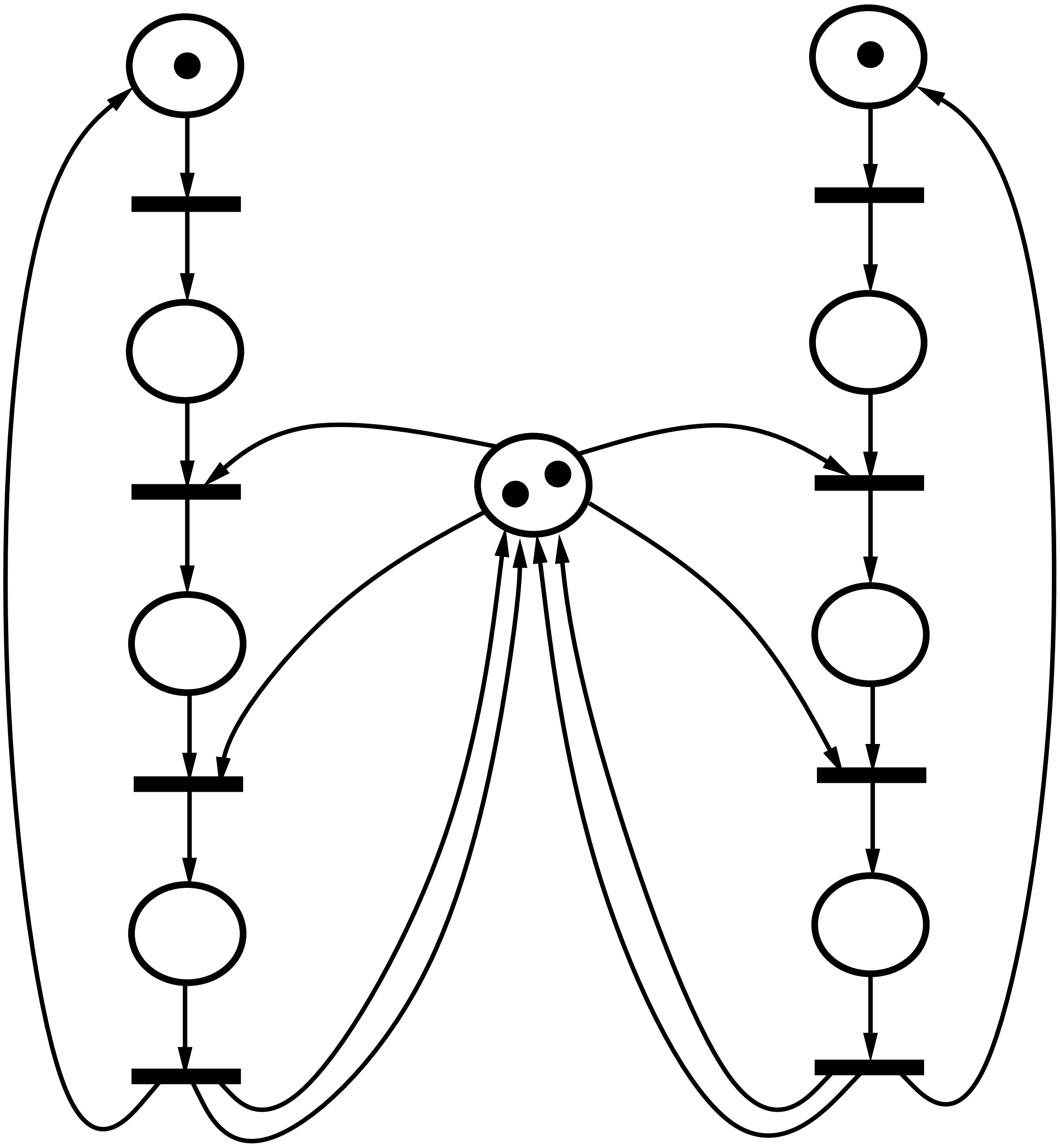
A Petri Net with a given marking is said to be in a **starvation** situation if and only if one or more transitions have been perminantly disabled (i.e., can never be enabled).

A Petri Net is **live** if every transition can eventually be fired. This is stronger that saying that the Petri Net cannot enter a deadlock situation.

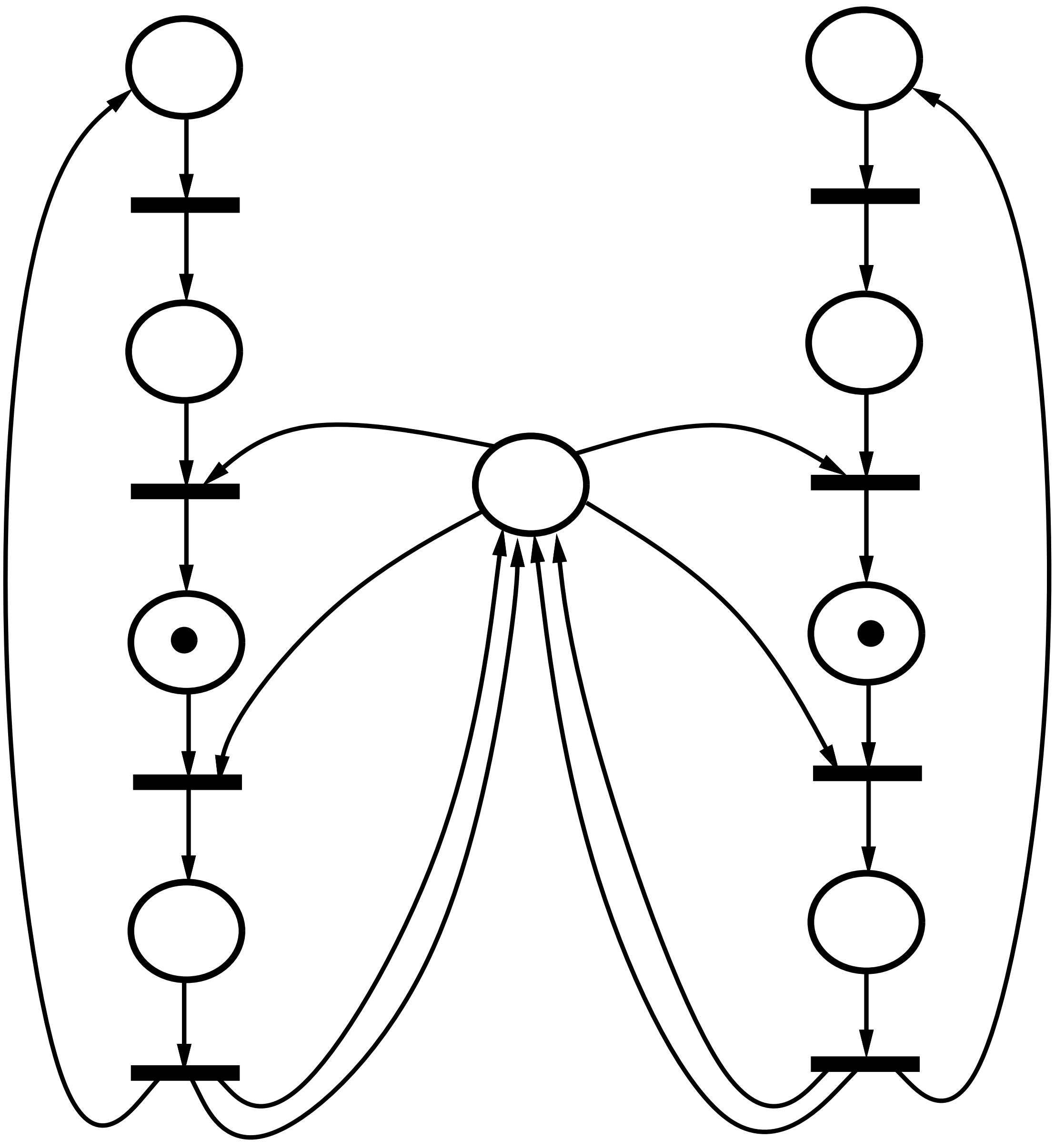
A Deadlocked Petri Net



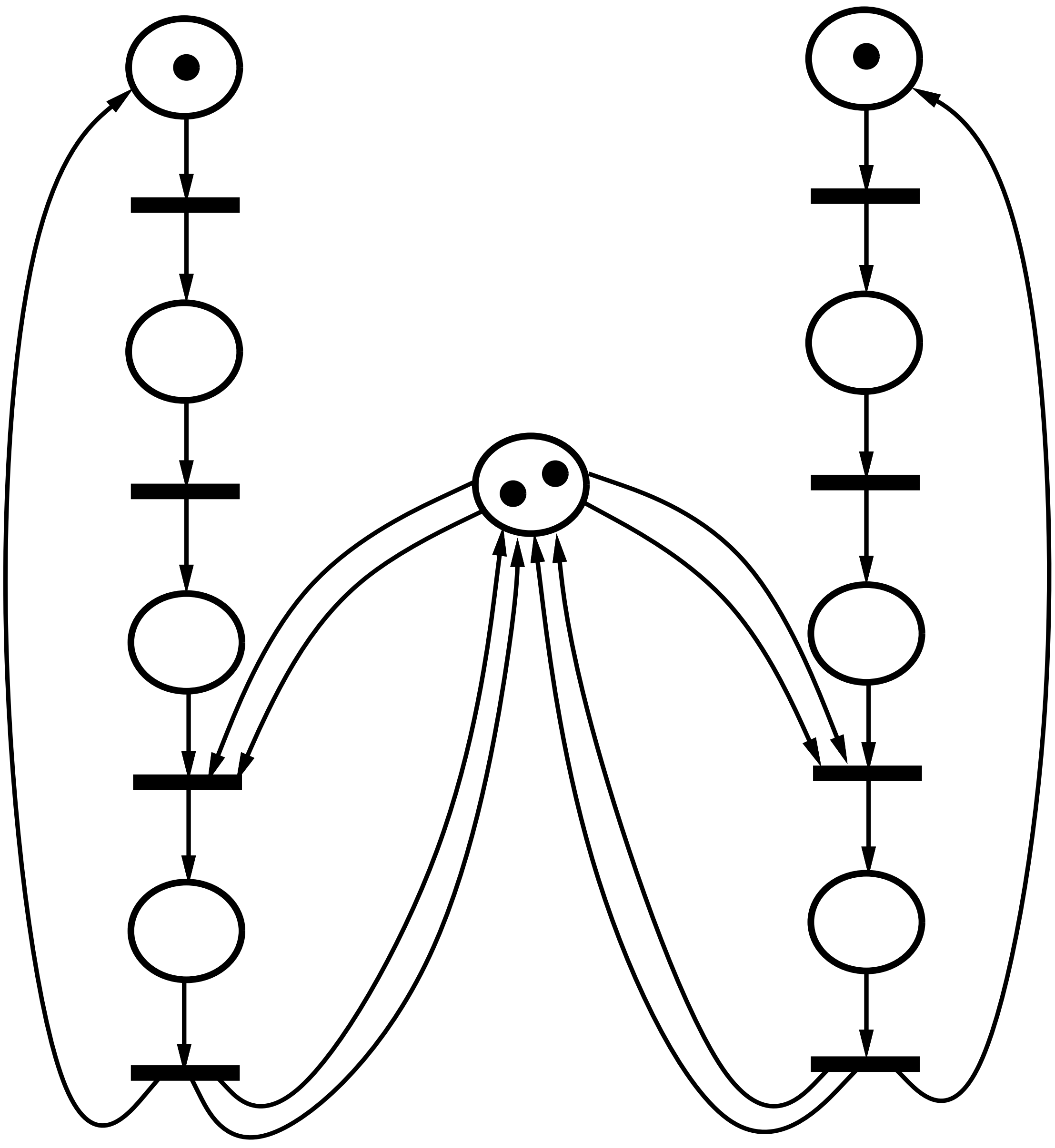
A Petri Net that can Enter a Deadlocked State



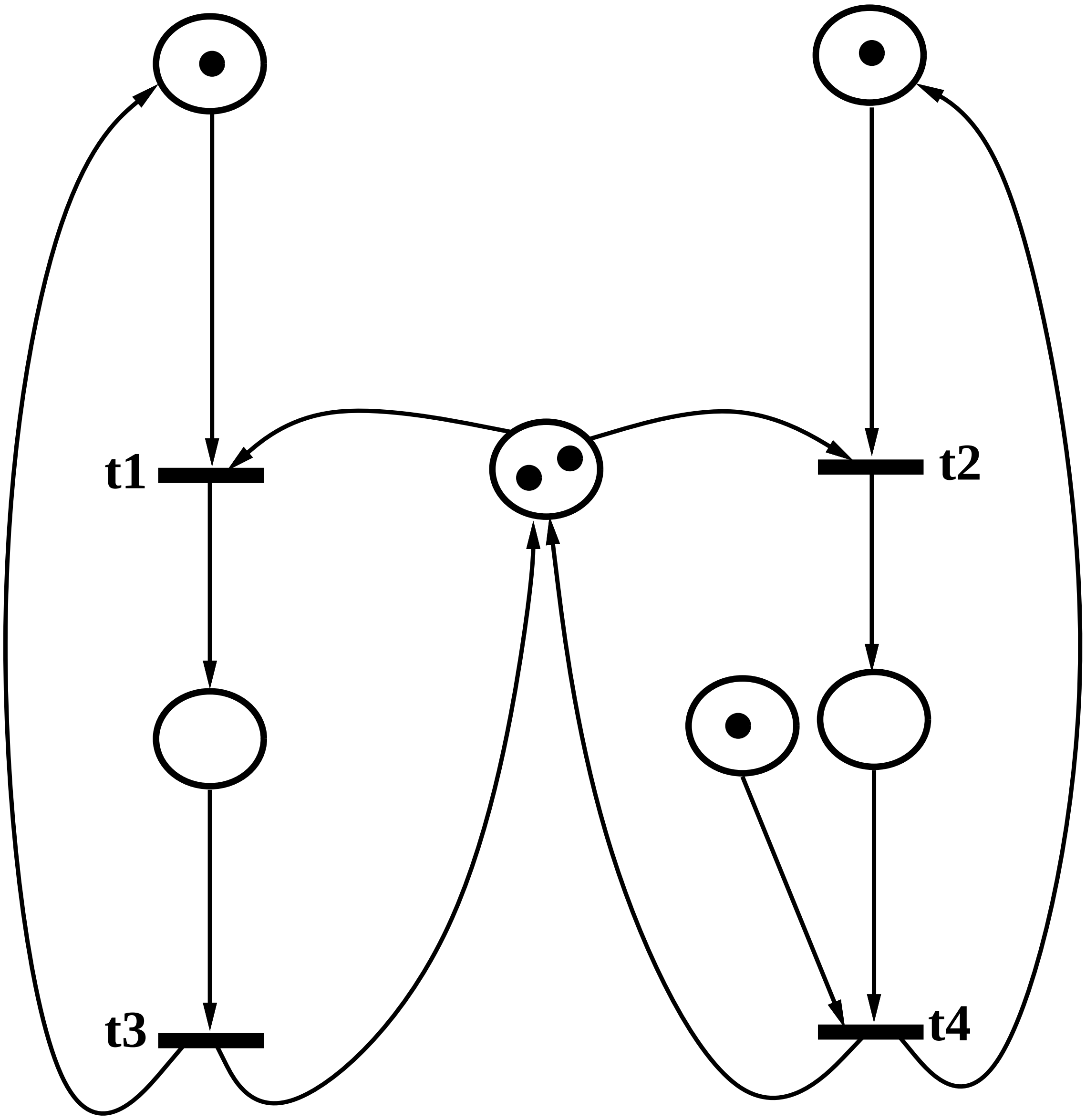
A Deadlocked Petri Net



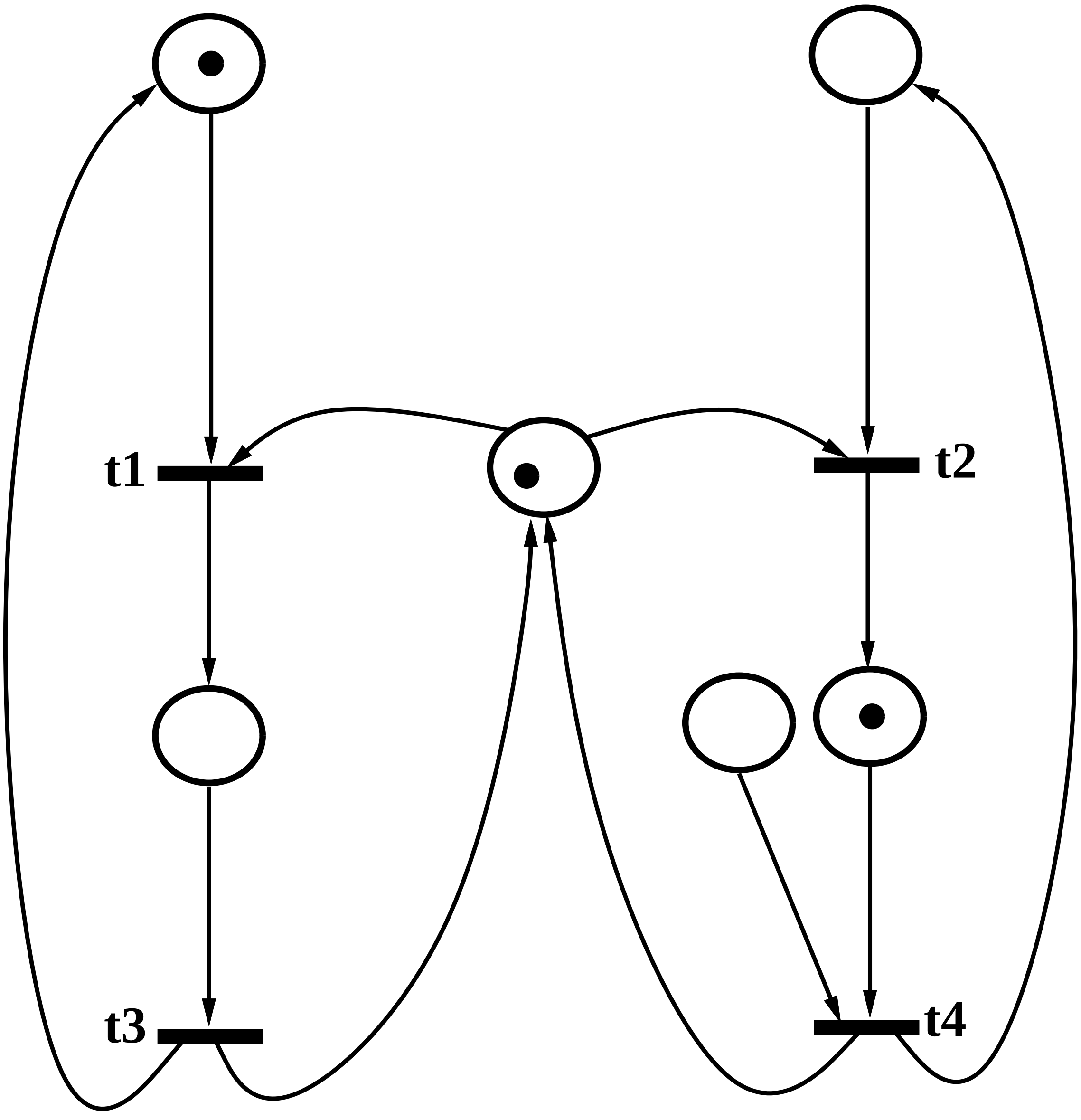
Modification into a Live Petri Net



A Petri Net, part of which which can go into Starvation



Starving Transitions t2 and t4



Shortcoming of Basic Petri Nets

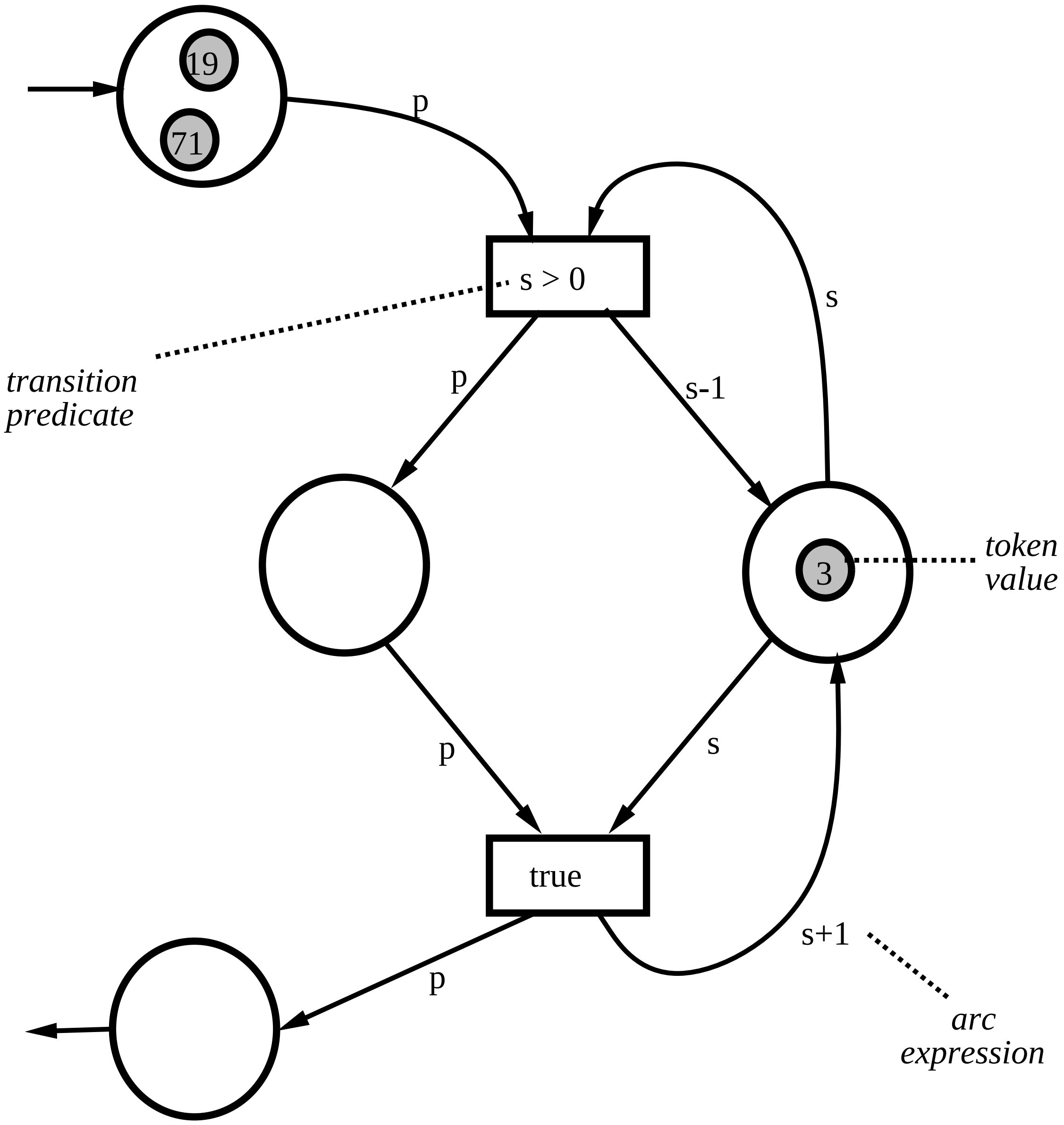
The Simplicity of building blocks leads to complexity in nets

For example, a semaphore of N processes requires $2N$ transitions and $3N+1$ places.

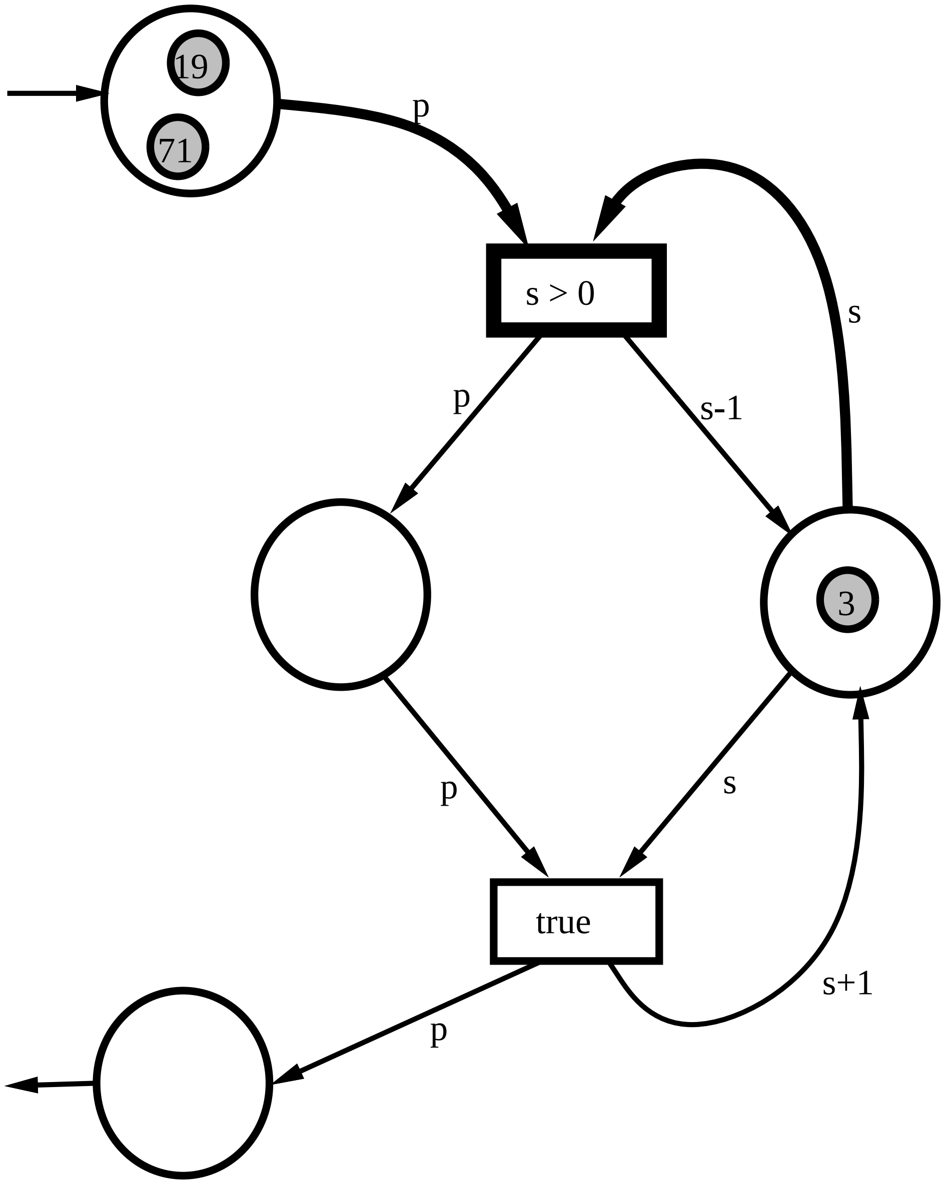
Would like:

- *Enable* and *fire* as computations
- *Tokens* as data, not just control
- Ability to reduce high-level Petri nets to basic Petri nets for analysis

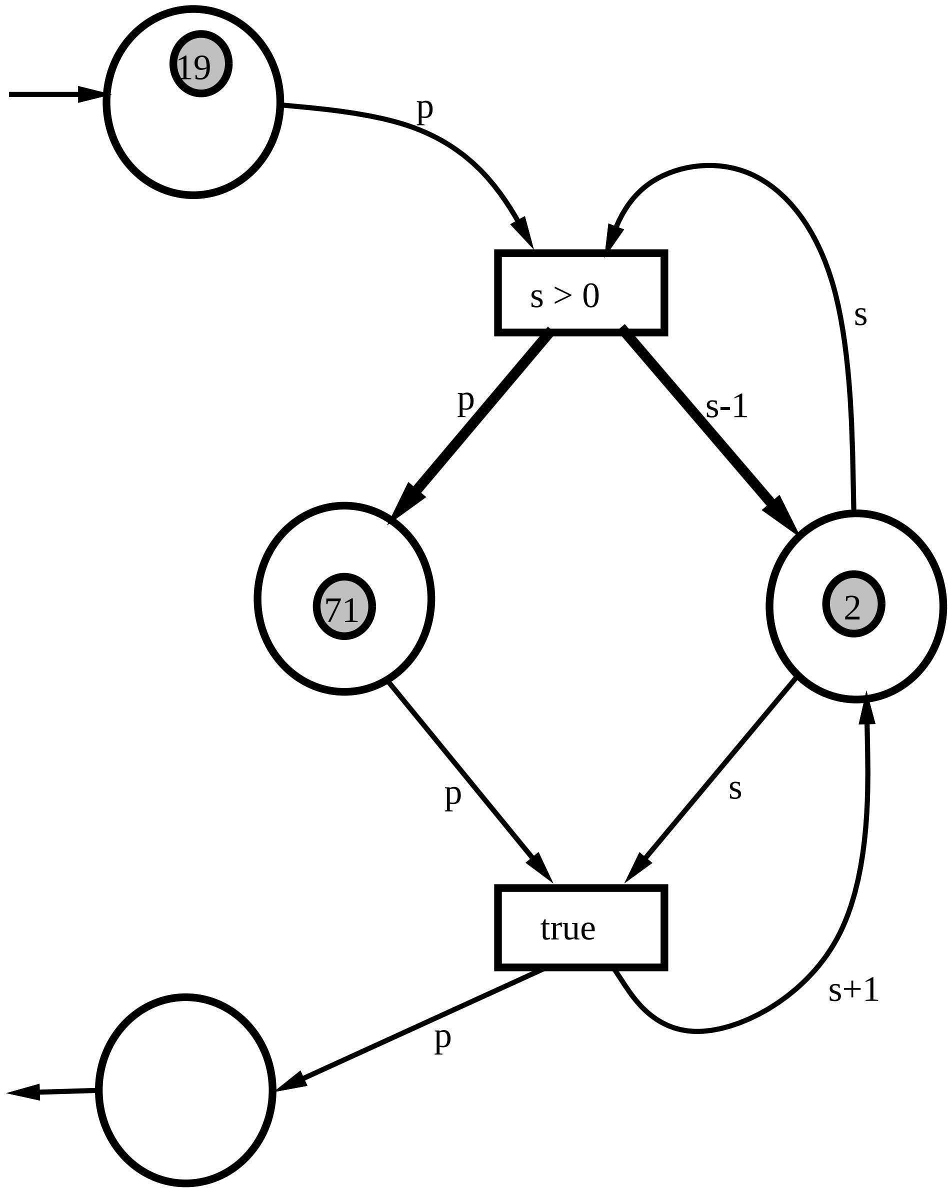
Higher-Level Net Semaphore



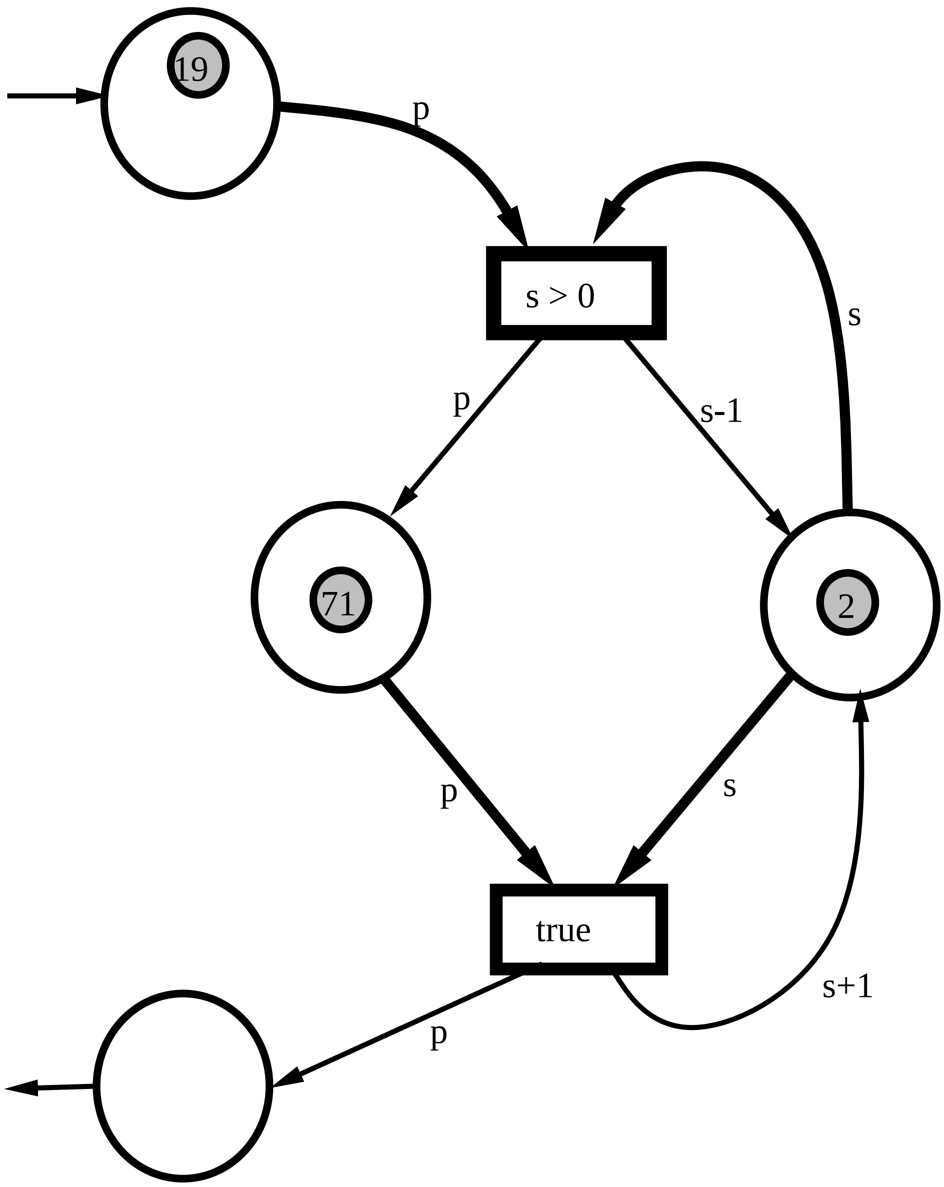
Enabled Transition



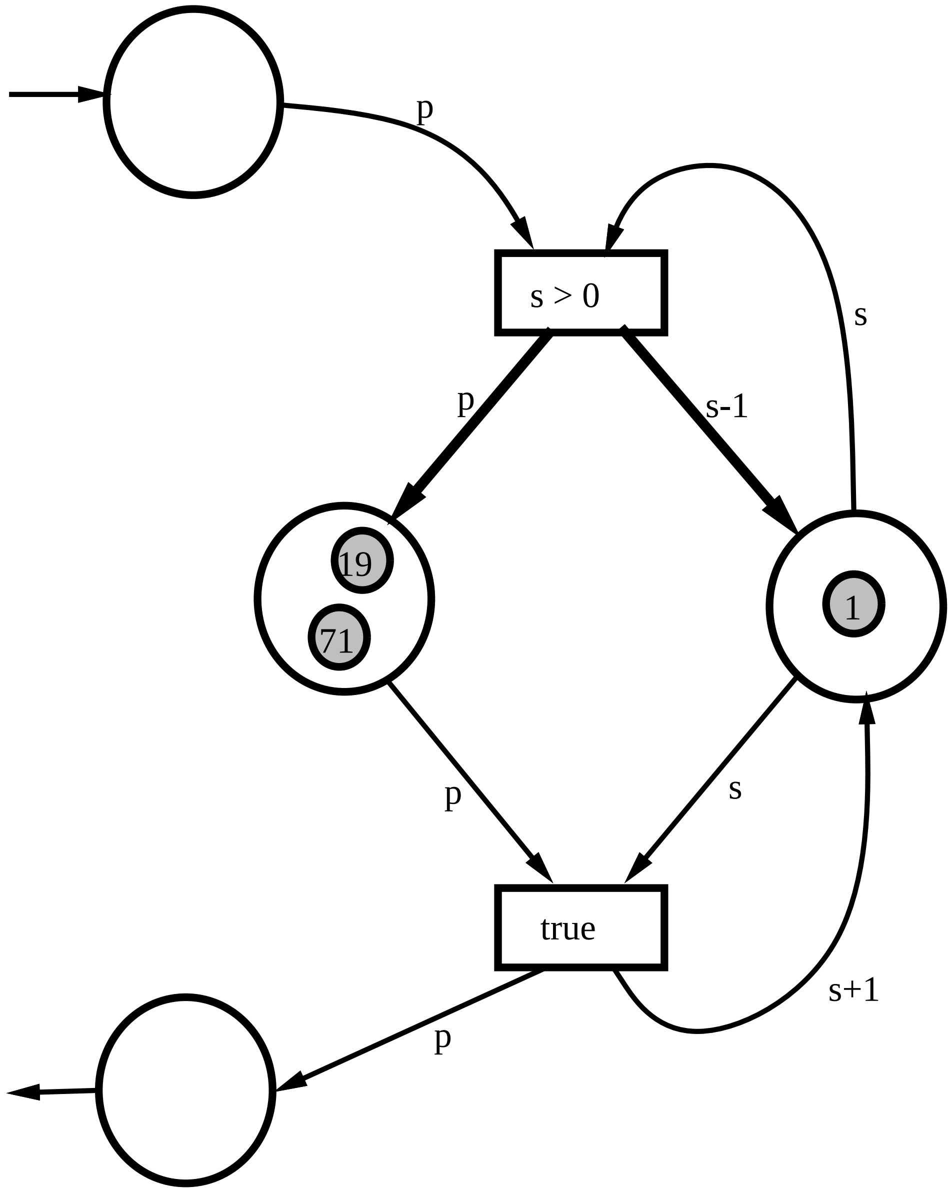
After Firing



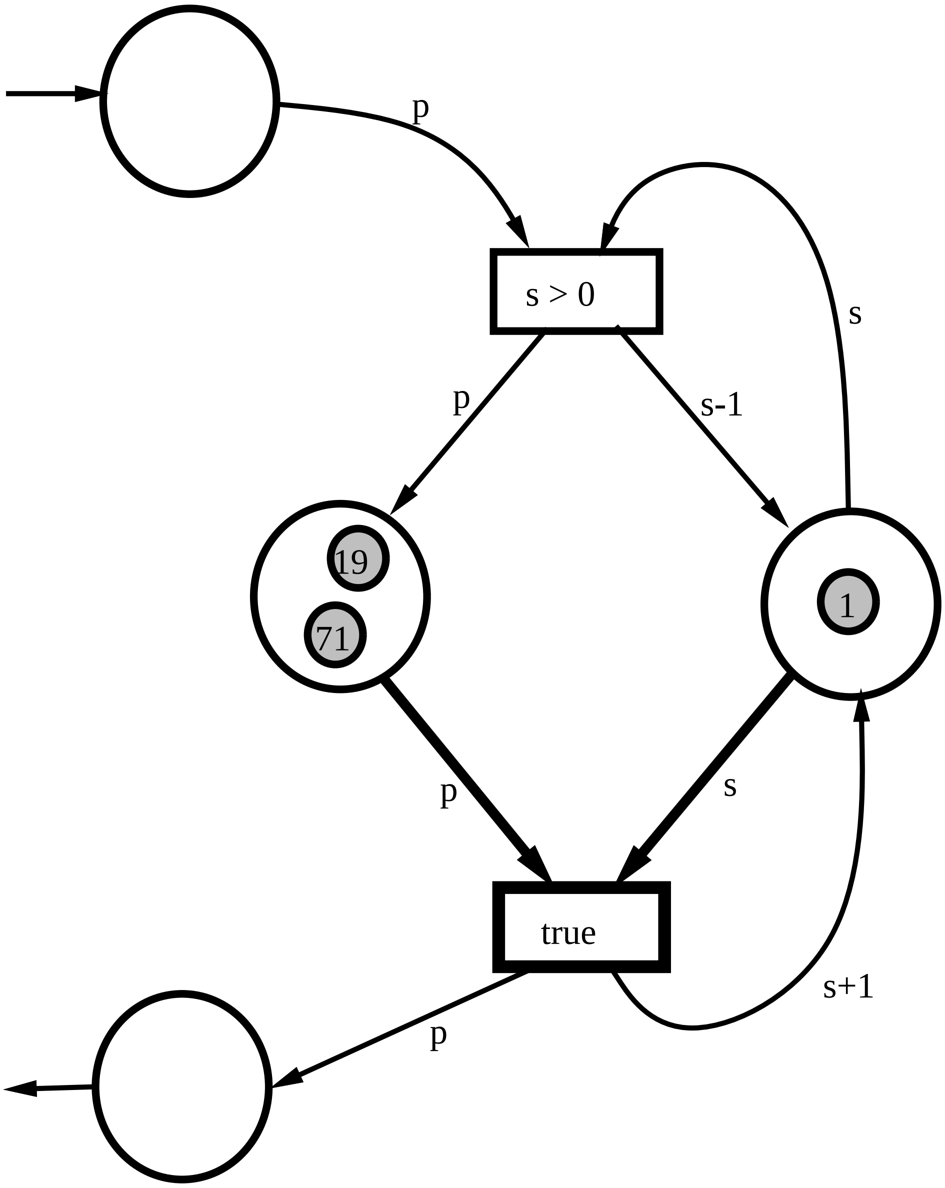
Enabled Transitions



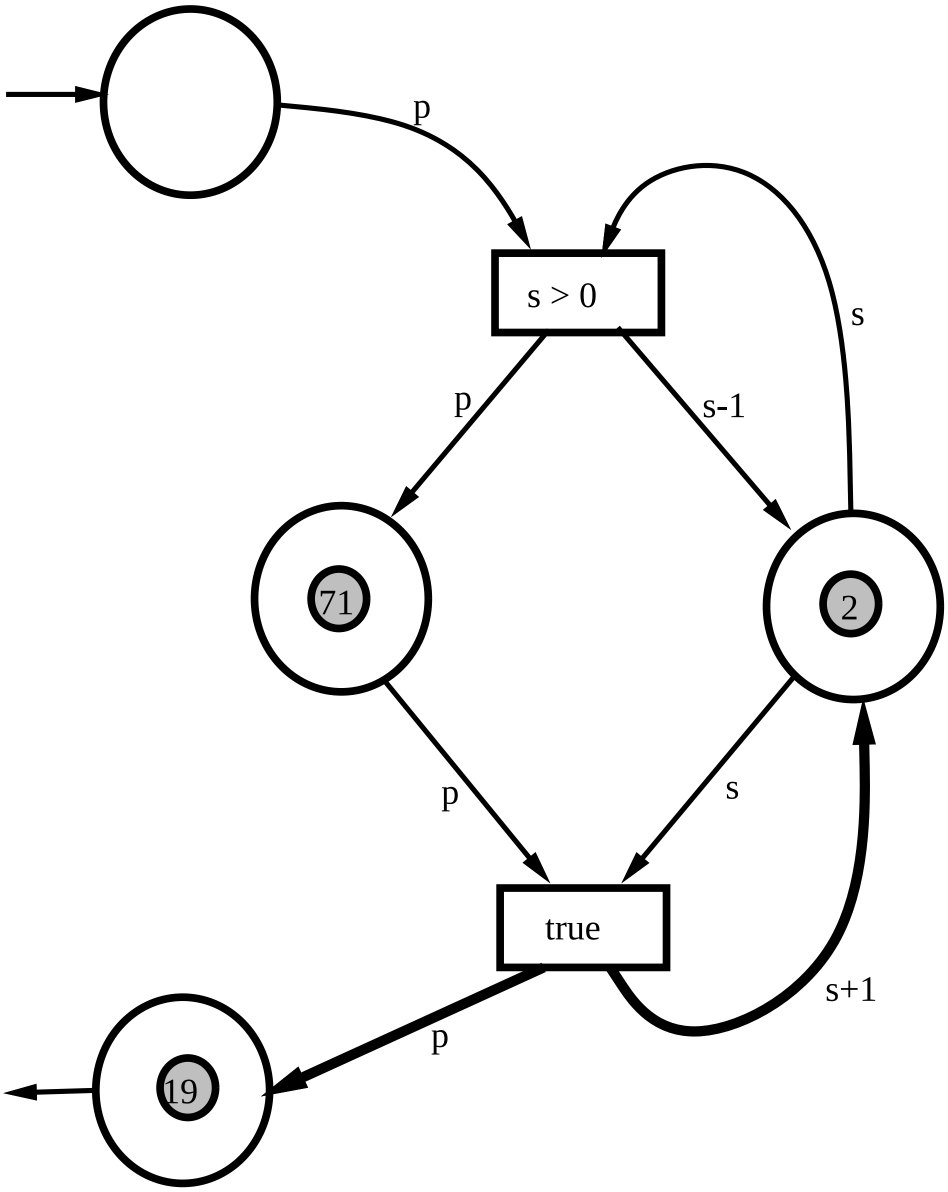
After Firing



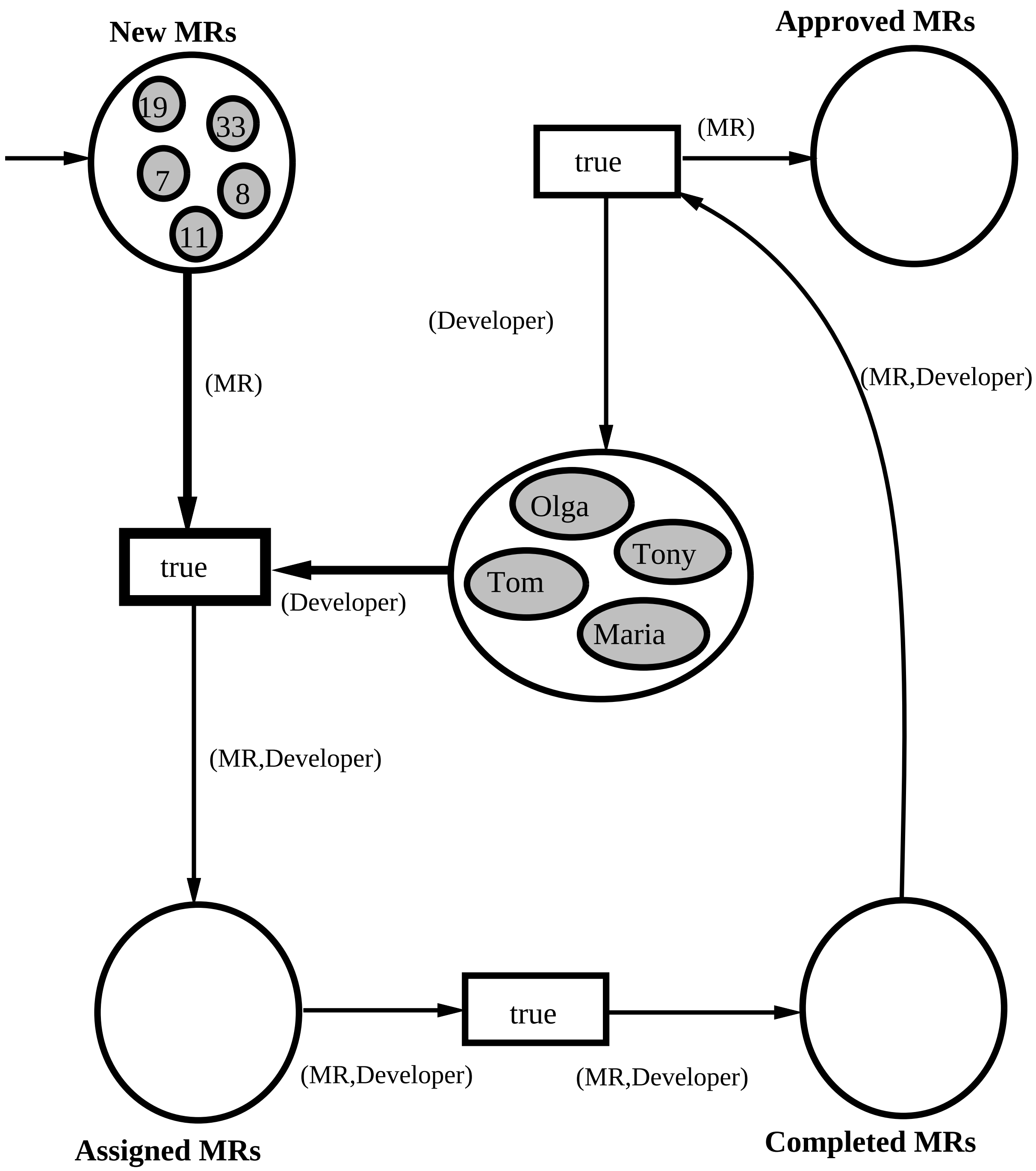
Enabled Transition



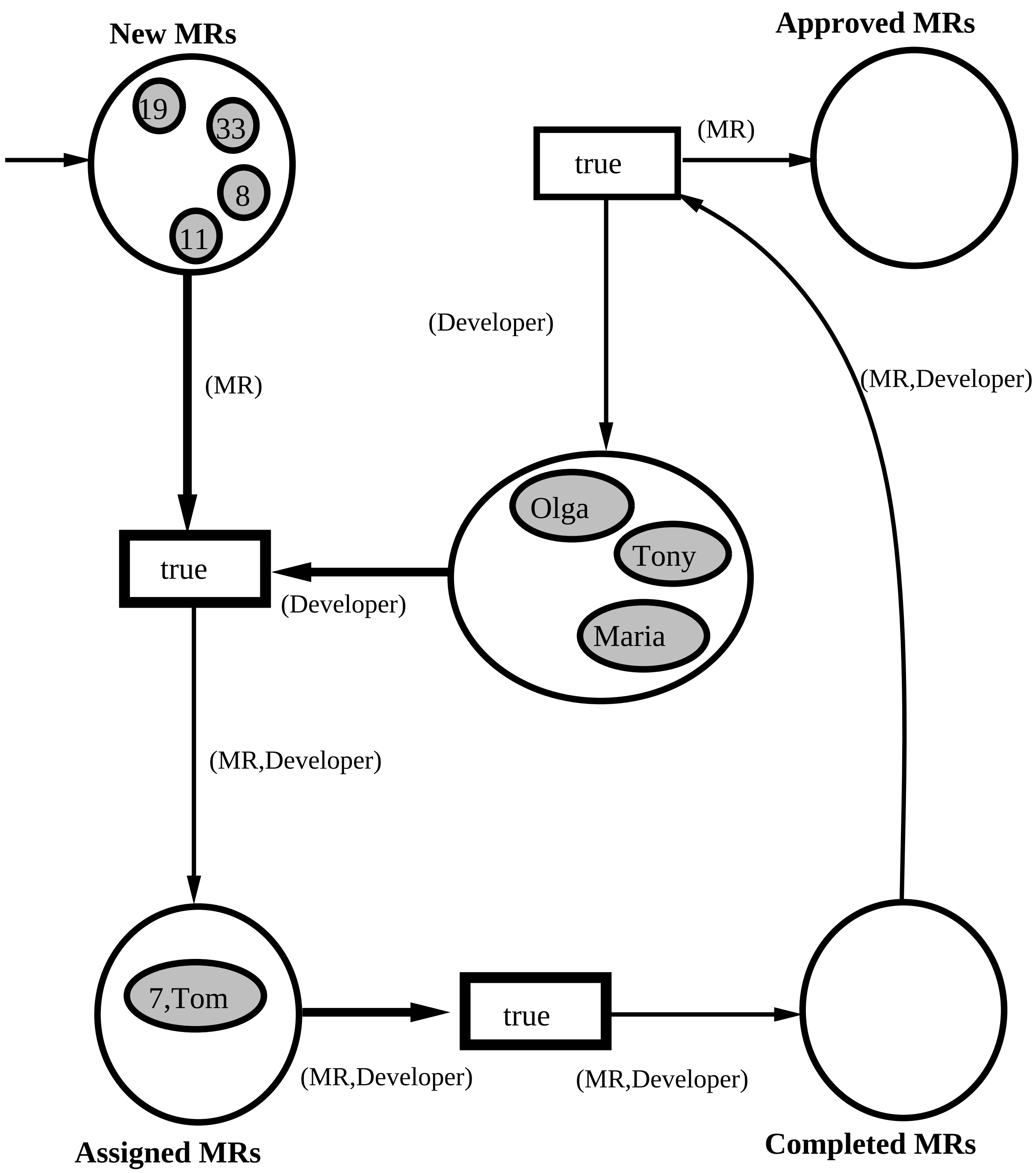
After Firing



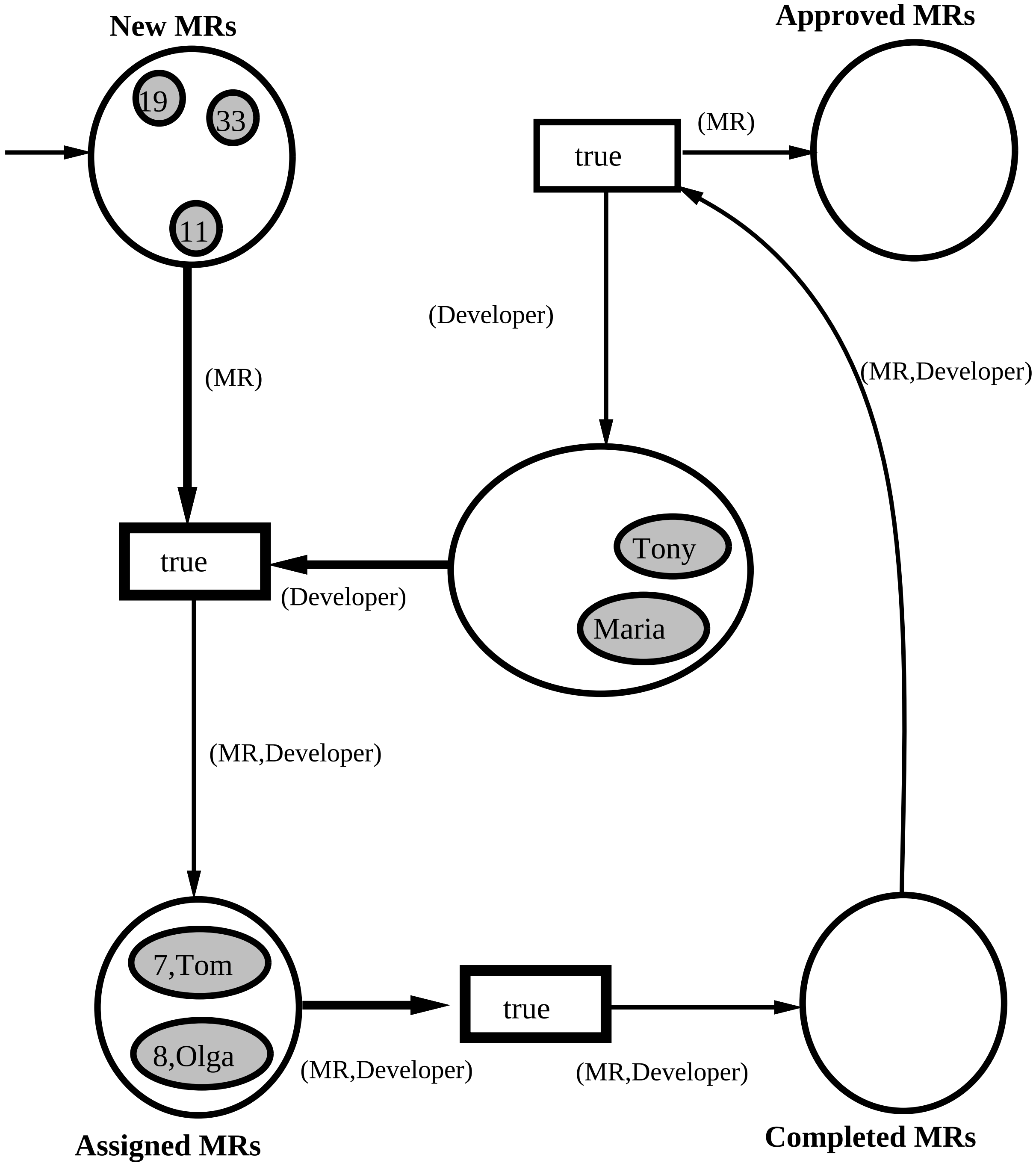
A Software Change Process



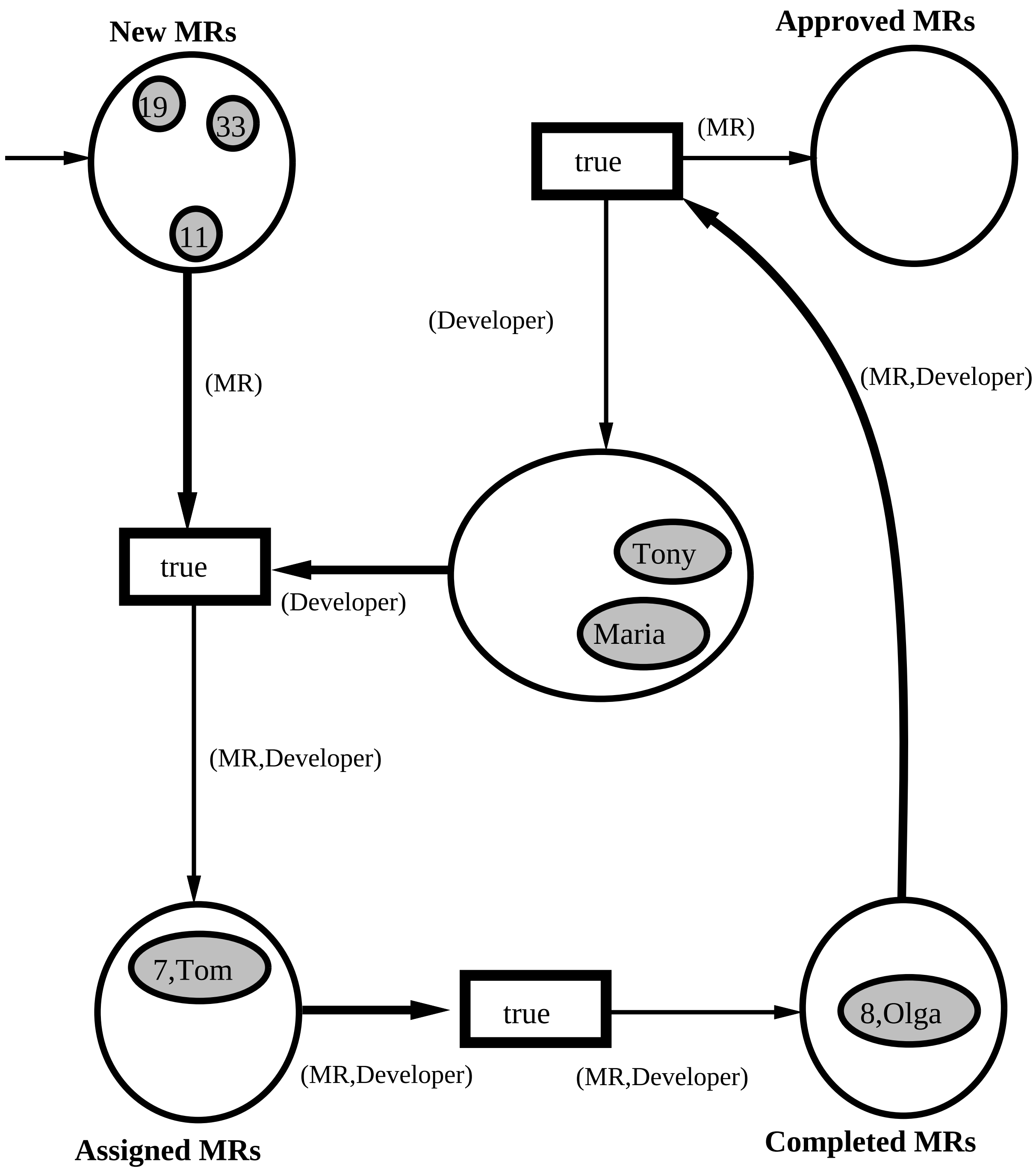
After Firing (New Assigned MR)



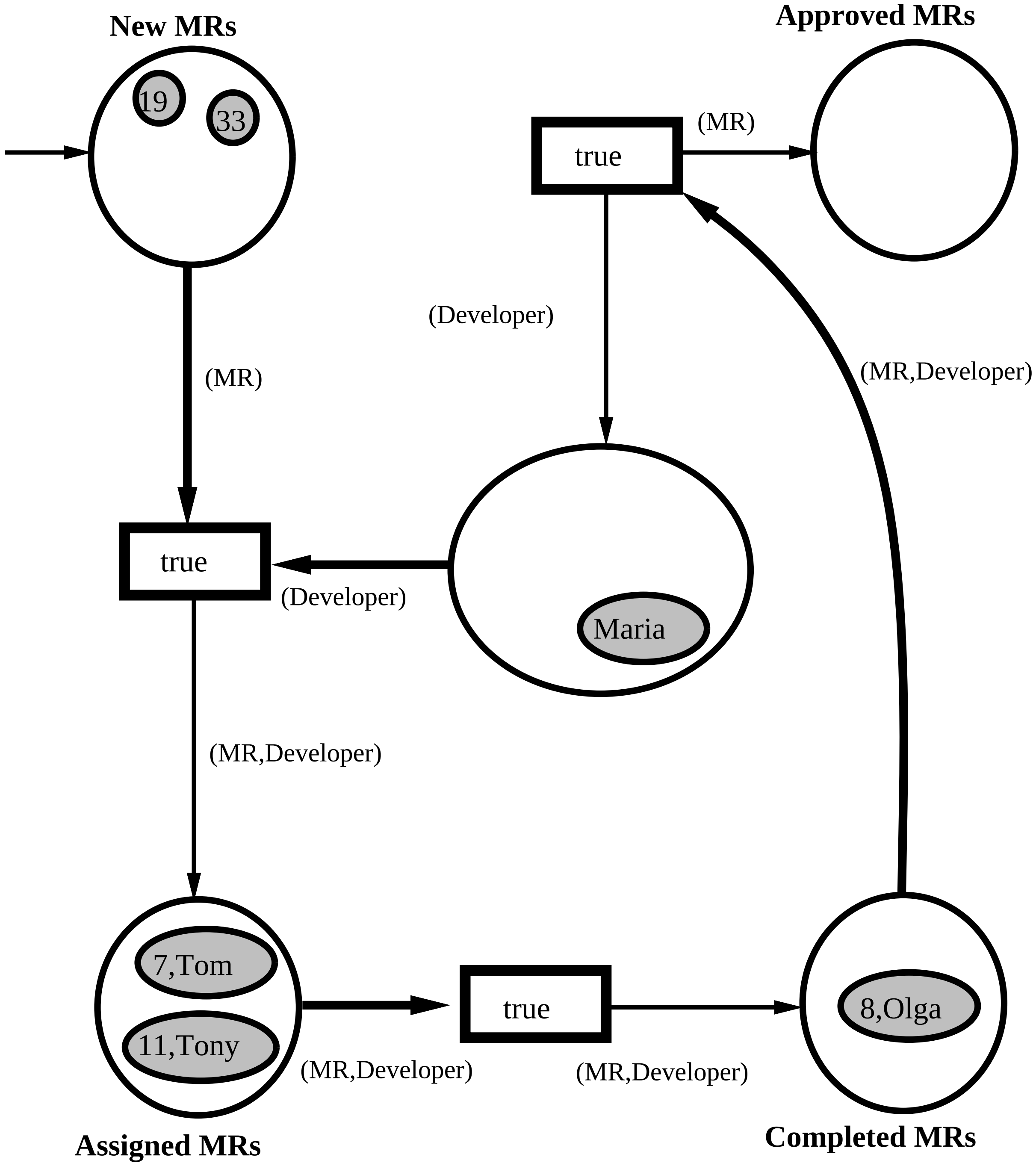
After Firing (New Assigned MR)



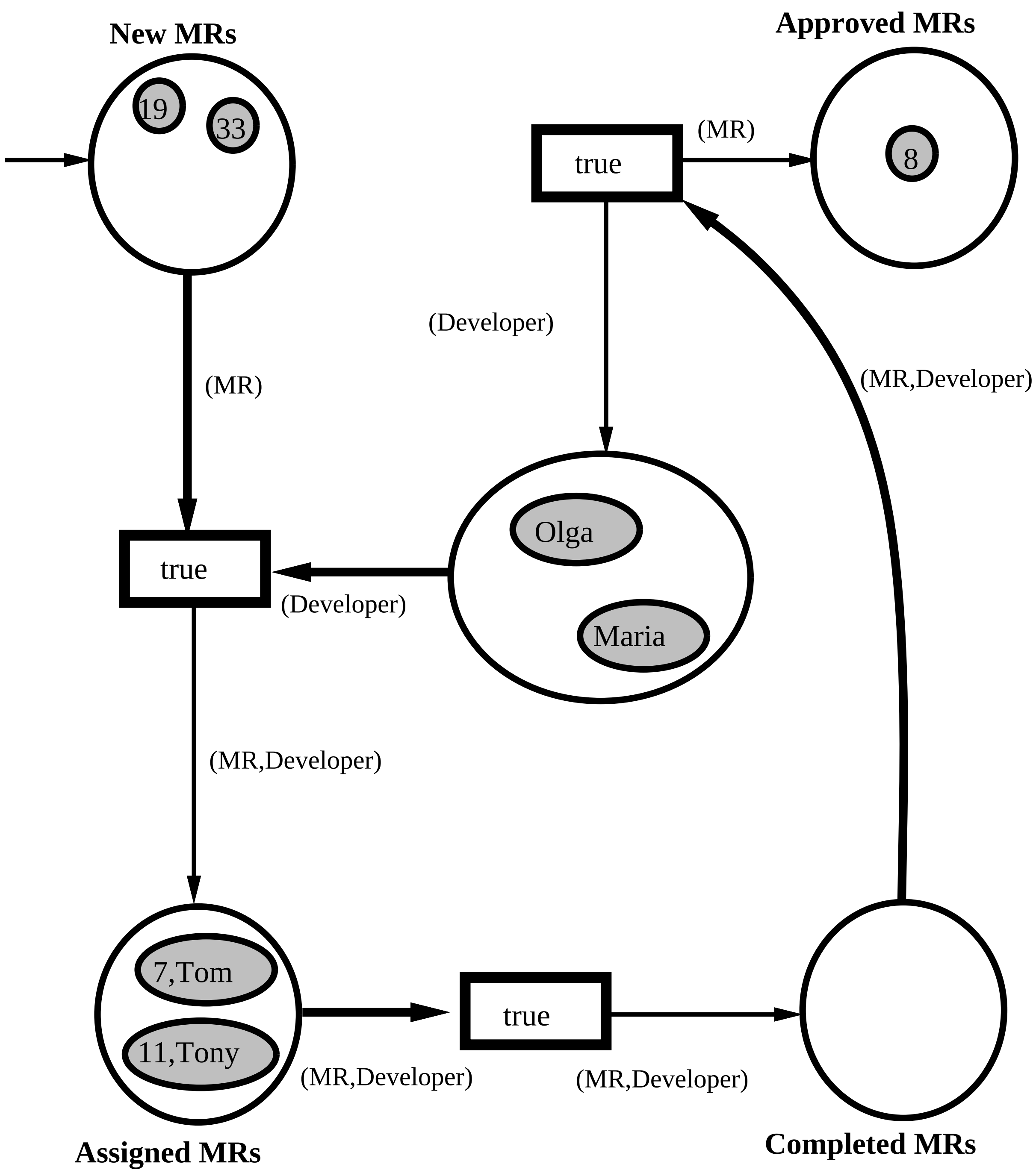
After Firing (New Completed MR)



After Firing (New Assigned MR)



After Firing (New Approved MR)



Design Patterns Topics

- Terminology and Motivation
- Reusable OO Design Patterns:
  - Adapter
  - Facade
  - Iterator
  - Composite
  - Template
  - Observer
  - Master-Slave

Terminology and Motivation

Design Patterns

- Good designers know not to solve every problem from first principles. They reuse solutions.
- Software practitioners do not do a good job of recording experience in software design for others to use.
- A **Design Pattern** systematically names, explains, and evaluates an important and recurring design.
- We describe a set of well-engineered design patterns that practitioners can apply when crafting their applications.
- Material drawn from [Gamma et al., 1995] and [Coplien and Schmidt, 1995].

Becoming a Master Designer

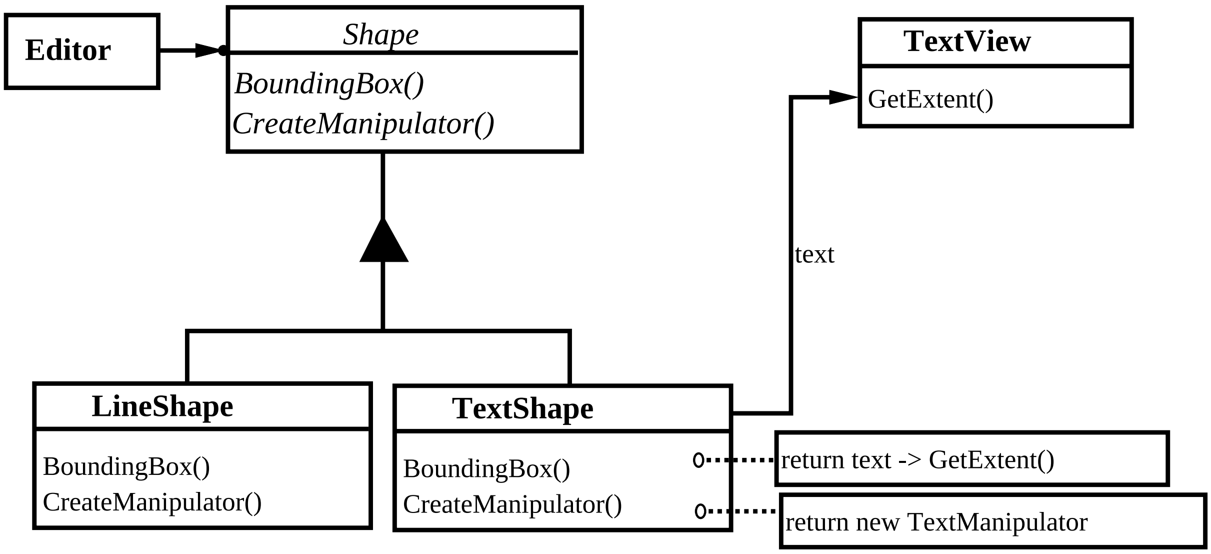
- **First, One Must Learn the Rules:**
  - Algorithms
  - Data Structures
  - Languages
- **Later, One Must Learn the Principles:**
  - Structured Programming
  - Modular Programming
  - OO Programming
- **Finally, One Must Study the Designs of Other Masters:**
  - Design patterns must be understood, memorized, and applied.
  - There are thousands of existing design patterns.

Reusable OO Design Patterns

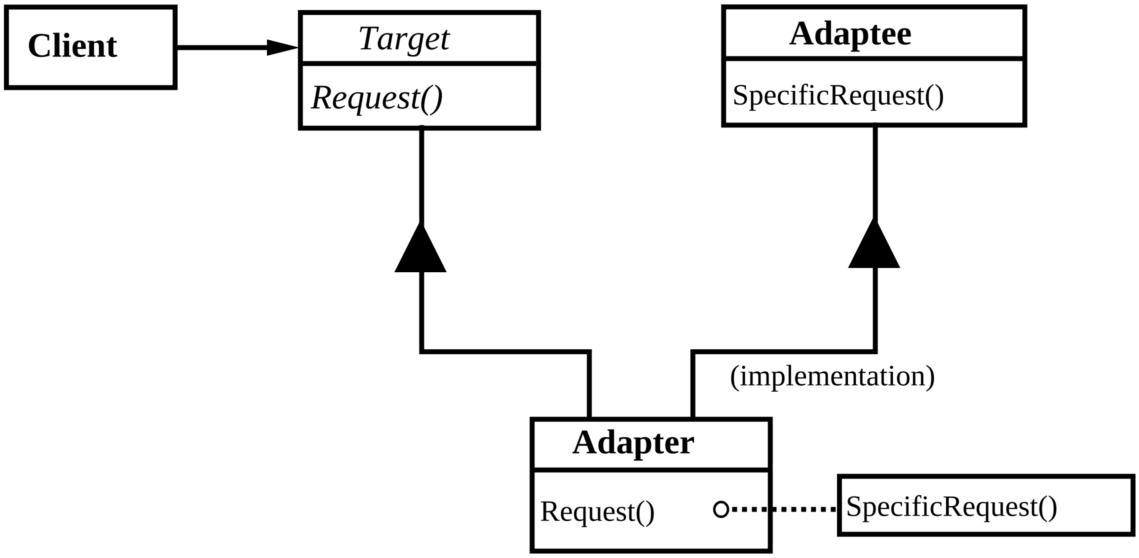
The Adapter Pattern

- **Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Motivation:** When we want to reuse classes in an application that expects classes with a different and incompatible interface, we do not want to (and often cannot) change the reusable classes to suit our application.

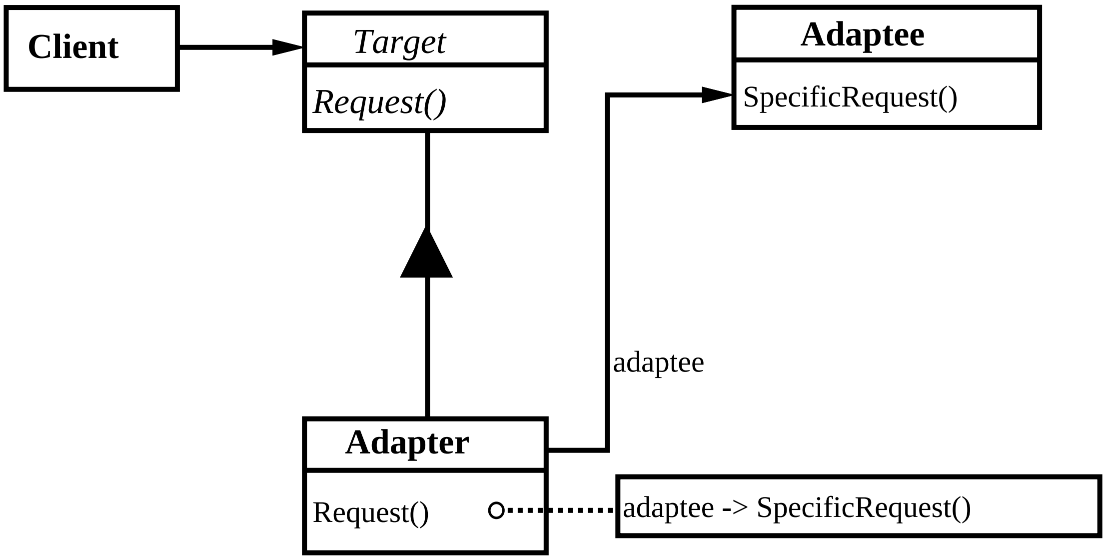
Example of the Adapter Pattern



Structure of the Adapter Pattern



Using Multiple Inheritance



Using Object Composition

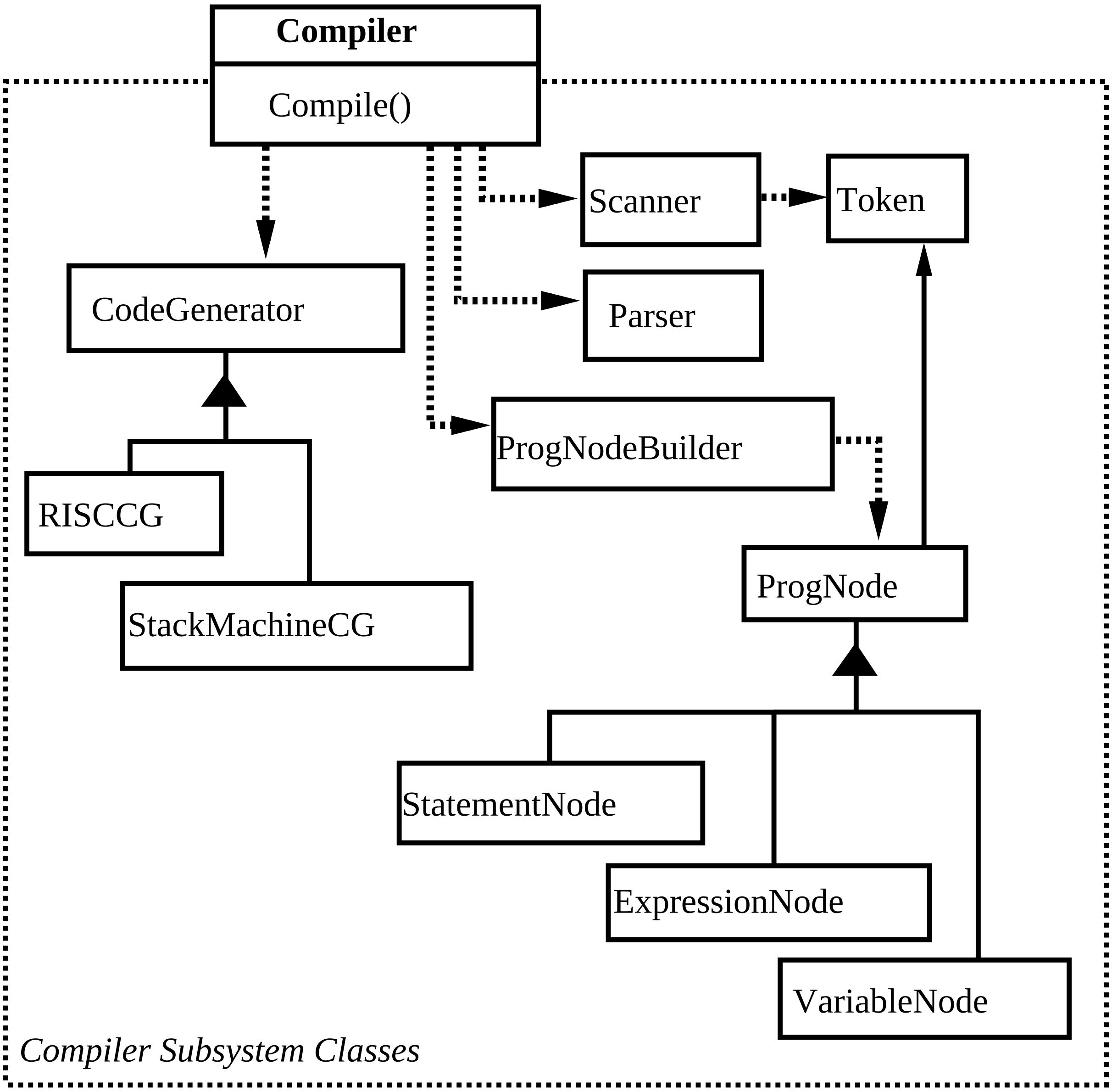
Participants of the Adapter Pattern

- **Target:** Defines the application-specific interface that clients use.
- **Client:** Collaborates with objects conforming to the target interface.
- **Adaptee:** Defines an existing interface that needs adapting.
- **Adapter:** Adapts the interface of the adaptee to the target interface.

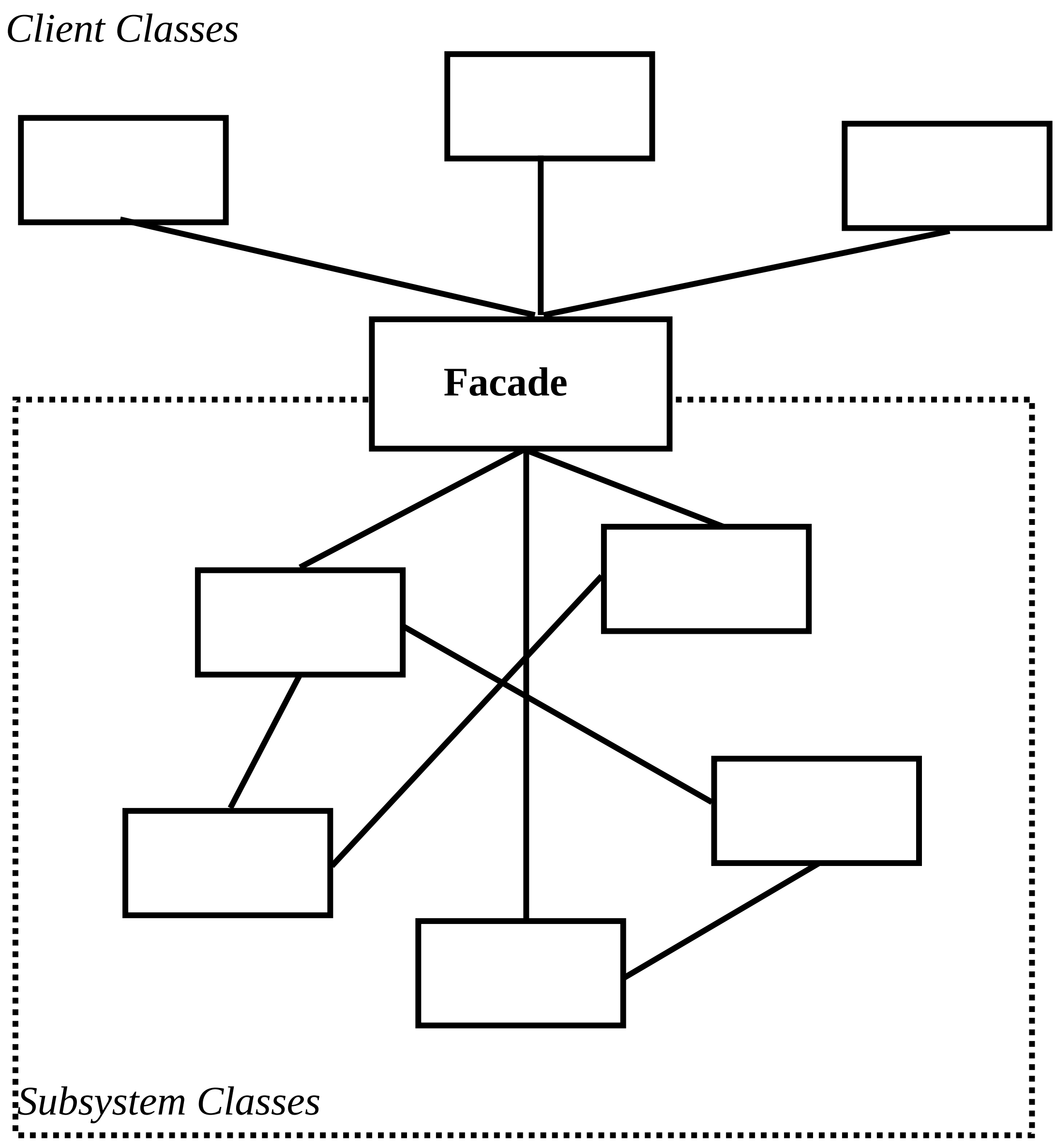
The Facade Pattern

- **Intent:** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Motivation:** Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to to so is to use a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

Example of the Facade Pattern



Structure of the Facade Pattern



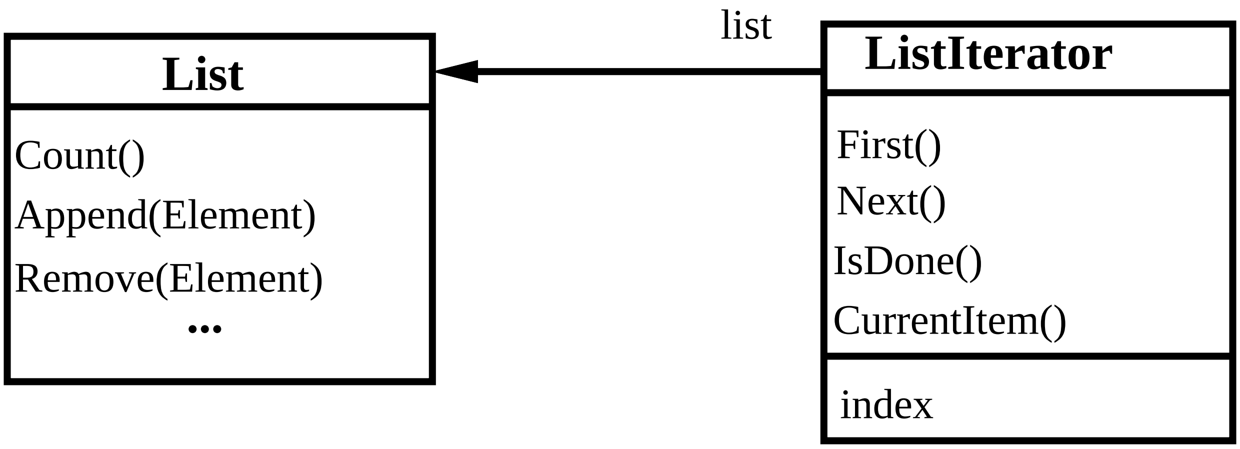
Participants of the Facade Pattern

- **Facade:**
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- **Subsystem Classes:**
  - Implement subsystem functionality.
  - Handle work assigned by the facade object.
  - Have no knowledge of the facade; that is, they keep no references to it.

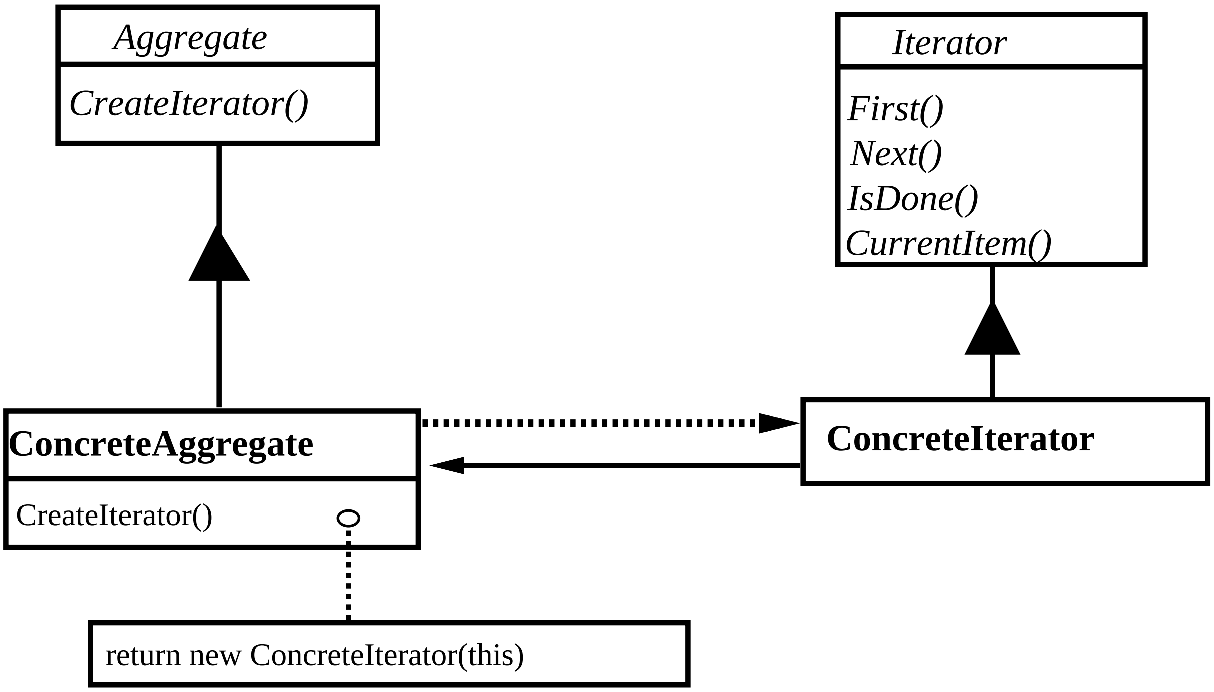
The Iterator Pattern

- **Intent:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Also to move the responsibility for access and traversal from the aggregate object to the iterator object.
- **Motivation:**
  - One might want to traverse an aggregate object in different ways.
  - One might want to have more than one traversal pending on the same aggregate object.
  - Not all types of traversals can be anticipated a priori.
  - One should not bloat the interface of the aggregate object with all these traversals.

Example of the Iterator Pattern



Structure of the Iterator Pattern



Participants of the Iterator Pattern

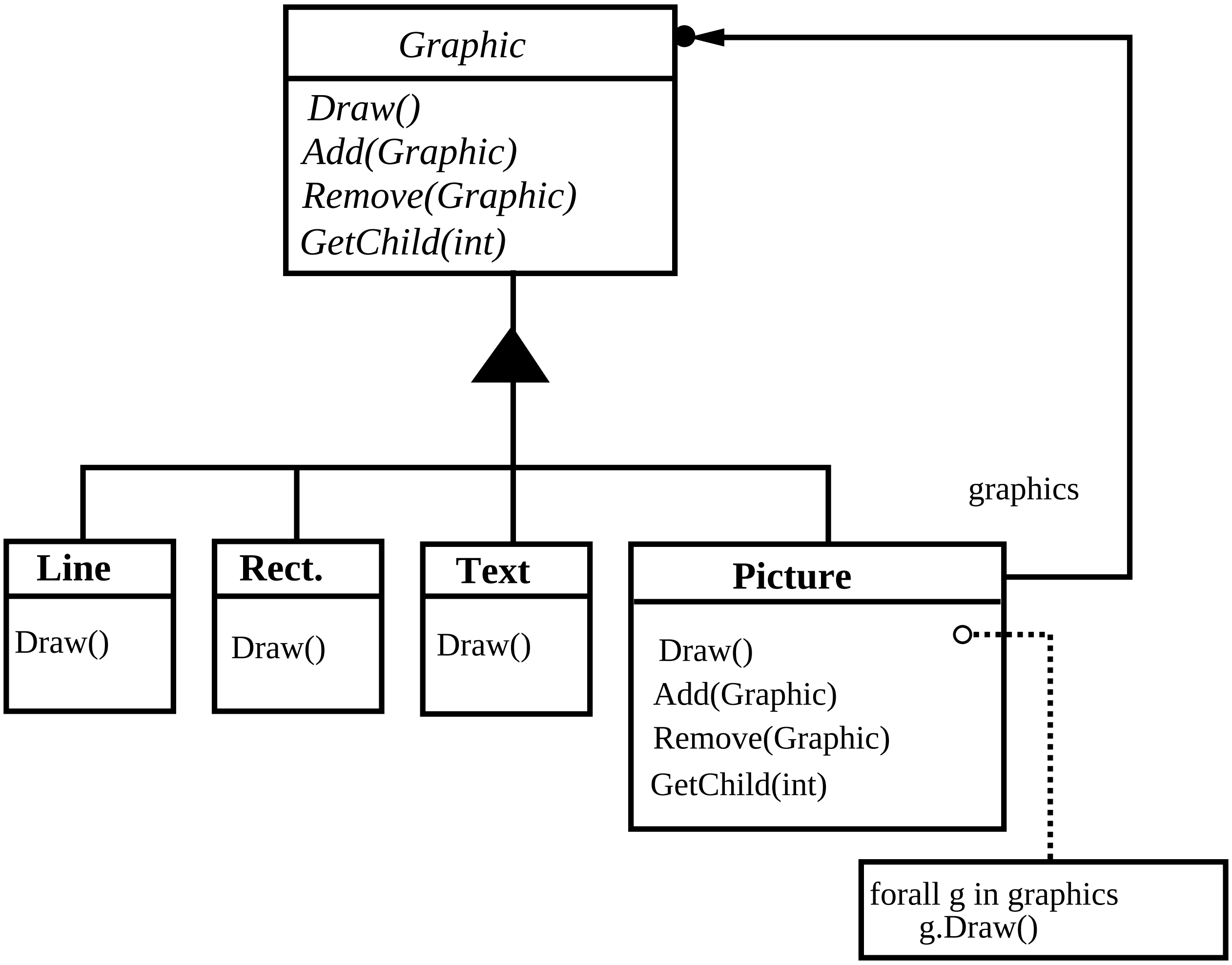
- **Iterator:** Defines an interface for accessing and traversing elements.
- **ConcreteIterator:**
  - Implements an iterator interface.
  - Keeps track of the current position in the traversal of the aggregate.
- **Aggregate:** Defines an interface for creating an iterator object.
- **ConcreteAggregate:** Implements the iterator creation interface to return an instance of the proper concrete iterator.

The Composite Pattern

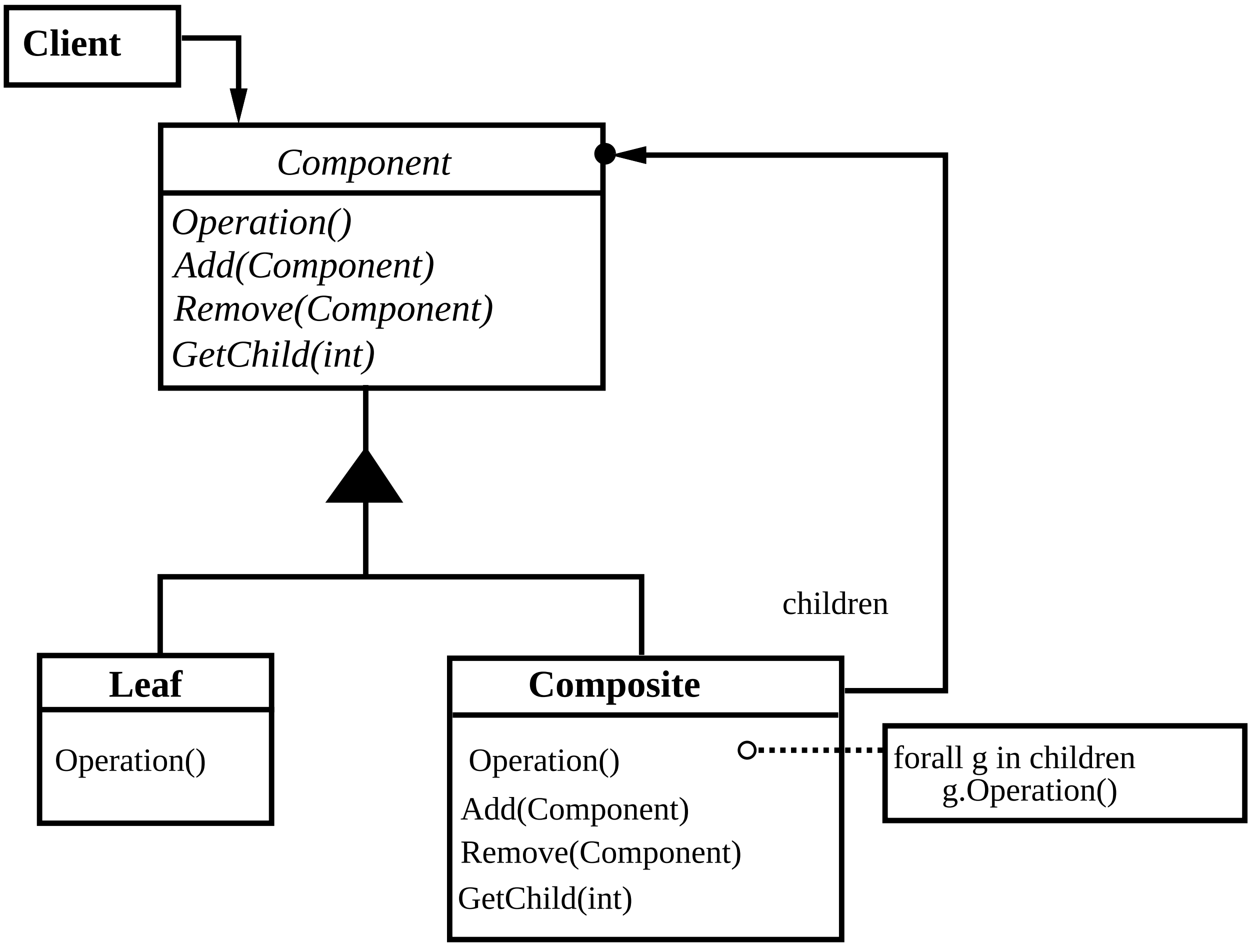
- **Intent:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- **Motivation:** If the composite pattern is not used, client code must treat primitive and container classes differently, making the application more complex than is necessary.

Example of the Composite Pattern



Structure of the Composite Pattern



Participants of Composite Pattern

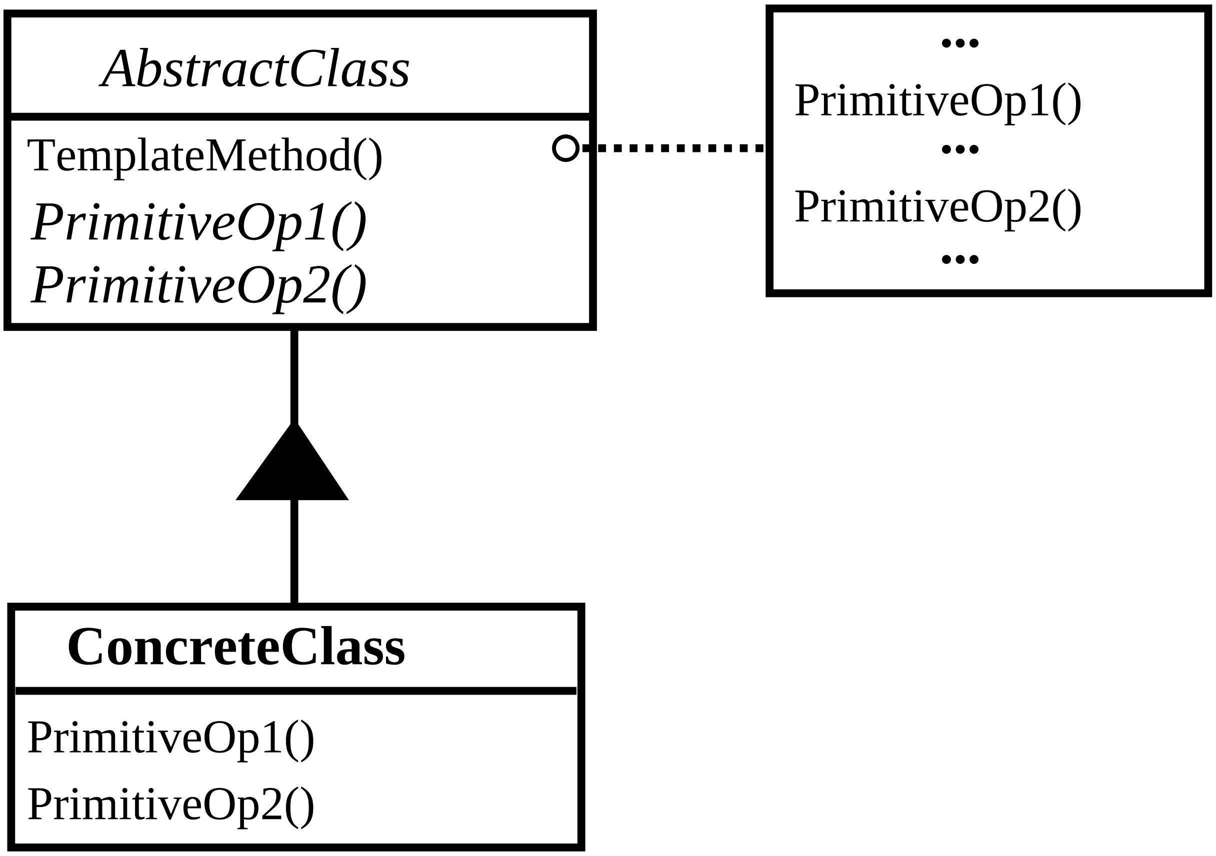
- **Component:**
  - Declares the interface for objects in the composition.
  - Implements default behavior for the interface common to all classes.
  - Declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure.
- **Leaf:**
  - Represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.

- **Composite:**
  - Defines behavior for components having children.
  - Stores child components.
  - Implements child-related operations in the component interface.
- **Client:** Manipulates objects in the composition through the component interface.

The Template Pattern

- **Intent:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Motivation:** By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering.

Structure of the Template Pattern



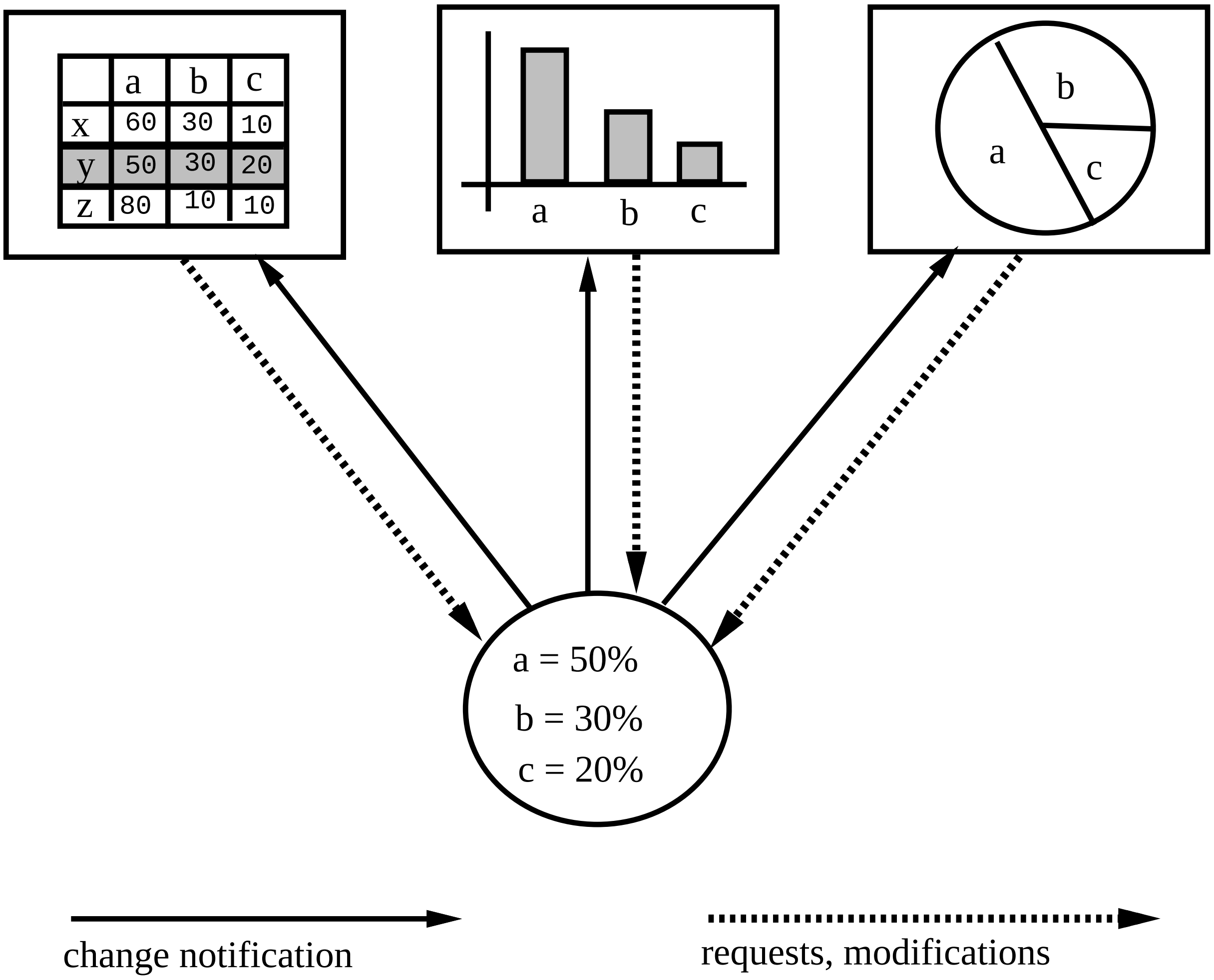
Participants of the Template Pattern

- **AbstractClass:**
  - Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
  - Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass:** Implements the primitive operations to carry out subclass-specific steps to the algorithm.

The Observer Pattern

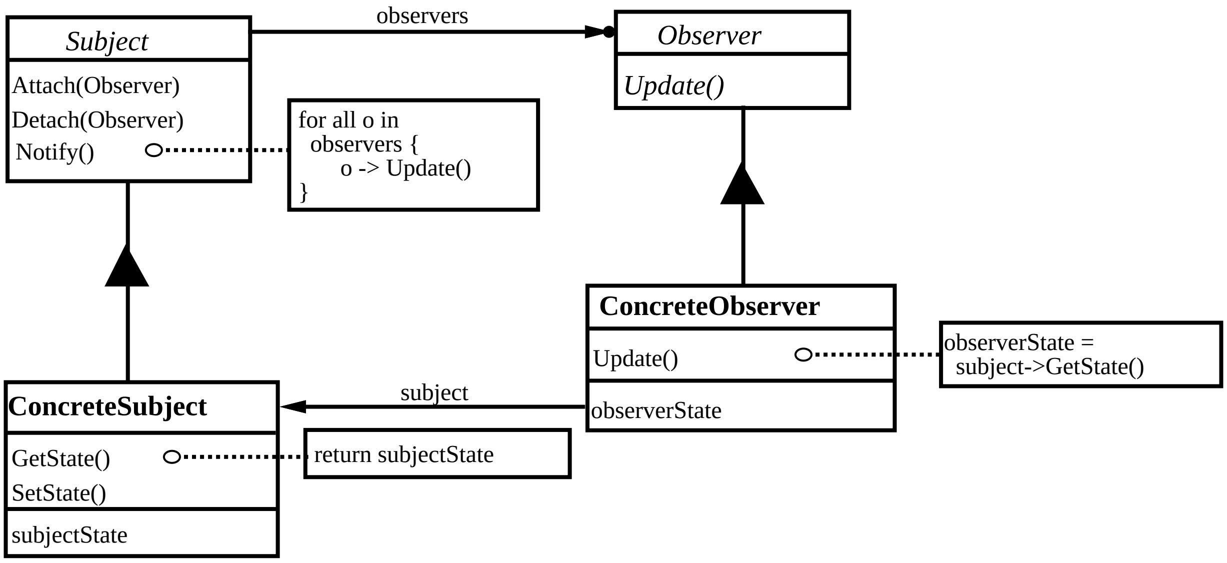
- **Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Motivation:** A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

Example of the Observer Pattern



Structure of the Observer Pattern

- The key objects in this pattern are **subject** and **observer**.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state.



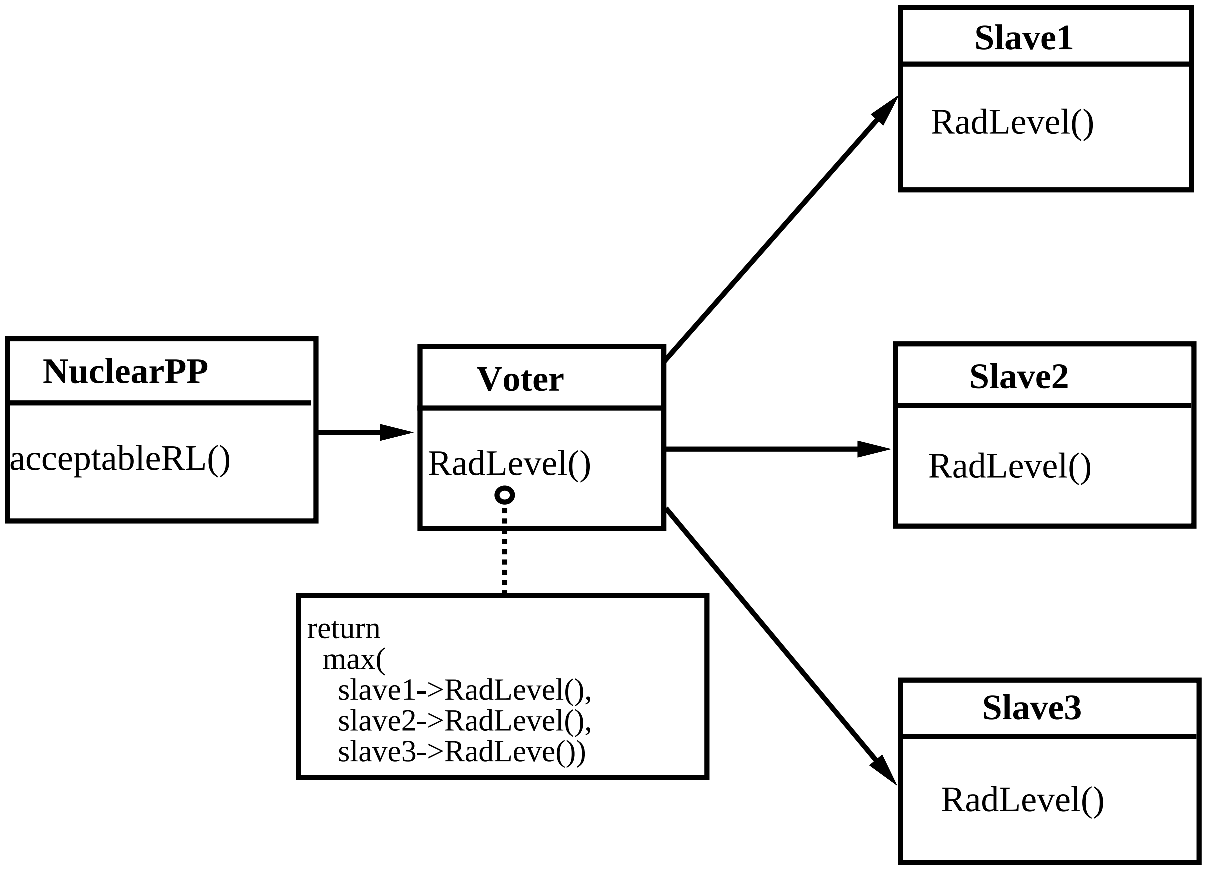
Participants of the Observer Pattern

- **Subject:**
  - Knows its numerous observers.
  - Provides an interface for attaching and detaching observer objects.
  - Sends a notification to its observers when its state changes.
- **Observer:** Defines an updating interface for concrete observers.
- **ConcreteSubject:** Stores state of interest to concrete observers.
- **ConcreteObserver:**
  - Maintains a reference to a concrete subject object.
  - Stores state that should stay consistent with the subject's.
  - Implements the updating interface.

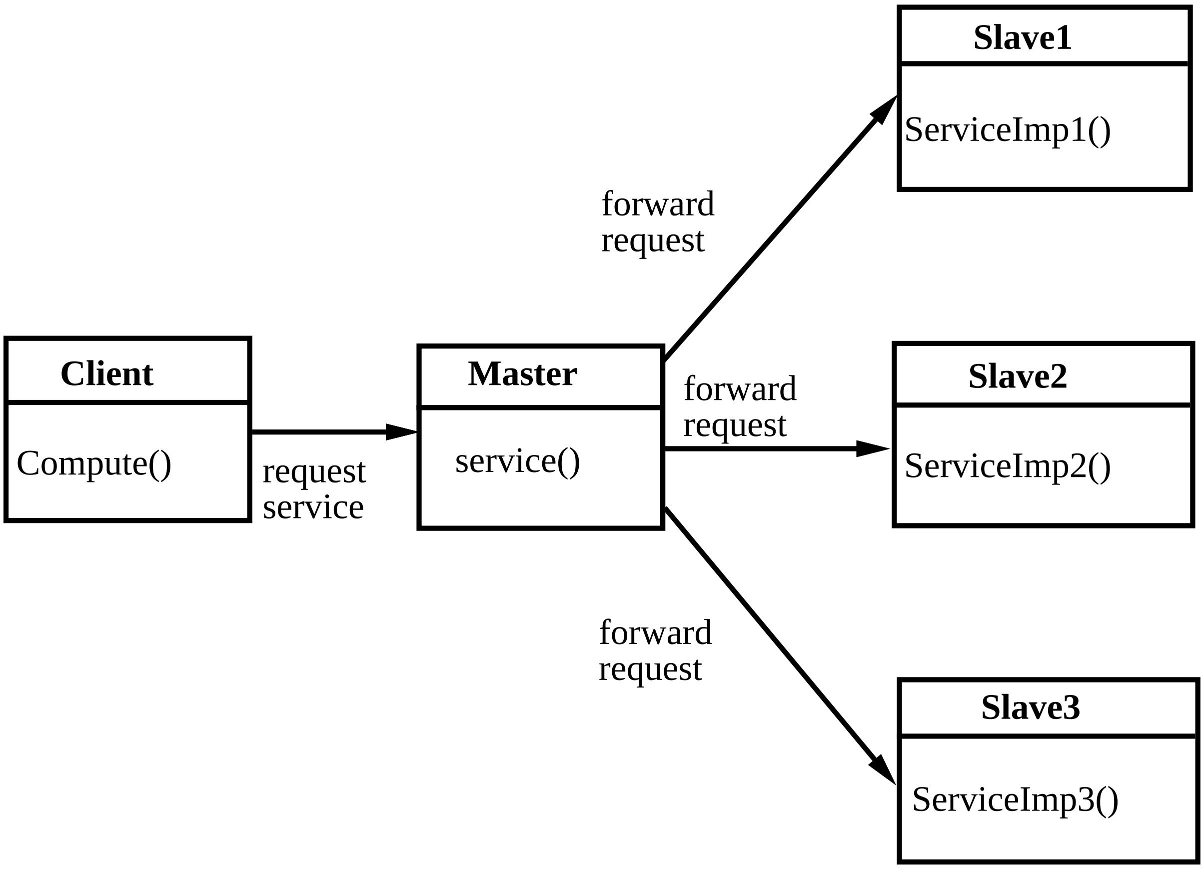
The Master-Slave (M/S) Pattern

- **Intent:** Handles the computation of replicated services within a software system to achieve fault tolerance and robustness. Independent components providing the same service (slaves) are separated from a component (master) responsible for invoking them and for selecting a particular result from the results returned by the slaves.
- **Motivation:** Fault tolerance is a critical factor in many systems. Replication of services and delegation of the same task to several independent suppliers is a common strategy to handle such cases.

Example of the M/S Pattern



Structure of the M/S Pattern



Participants of the M/S Pattern

- **Slave:** Implements a service.
- **Master:**
  - Organizes the invocation of replicated services.
  - Decides which of the results returned by its slaves is to be passed to its clients.
- **Client:** requires a certain service in order to solve its own task.

References

[Budgen, 1994] Budgen, D. (1994). *Software Design.* Addison-Wesley Publishing Company Inc., Wokingham, England.

[Cardelli et al., 1992] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. (1992). Modula-3 Language Definition. *ACM SIGPLAN Notices*, 27(8):15–42.

[Cooprider, 1979] Cooprider, L. W. (1979). The Representation of Families of Software Systems. Technical Report CMU–CS–79–116, Computer Science Department Carnegie-Mellon University.

[Coplien and Schmidt, 1995] Coplien, J. O. and Schmidt, D. C. (1995). *Pattern Languages of Program Design.* Addison-Wesley Professional Computing Series, Reading, Massachusetts.

[DeRemer and Kron, 1976] DeRemer, F. and Kron, H. H. (1976). Programming in the Large Versus Programming in the Small. *IEEE Transactions on Software Engineering*, 2(2):80–86.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of*

Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Reading, Massachusetts.

[Garlan and Shaw, 1993] Garlan, D. and Shaw, M. (1993). *In Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company.

[Ghezzi et al., 1991] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). *Fundamentals of Software Engineering*. Prentice Hall International, Englewood Cliffs, New Jersey.

[Harel, 1988] Harel, D. (1988). On Visual Formalisms. *Communications of the ACM*, 31(5):514–530.

[Holt and West, 1994] Holt, R. C. and West, T. (1994). *Turing Reference Manual (5th ed.)*. Holt Software Associates Inc., Toronto, Ontario.

[Lamb and Schneider, 1992] Lamb, D. A. and Schneider, K. A. (1992). Formalization of Information Hiding Design Methods. Technical Report External 0836–0227–92–337, Deprartment of Computing and Information Science Queen's University.

[Mancoridis, 1996a] Mancoridis, S. (1996a). *Controlling the Interactions of Architectural Design Components Using Scoping Rules*. PhD thesis, Department of Computer Science, University of Toronto.

[Mancoridis, 1996b] Mancoridis, S. (1996b). Toward a Generic Framework for Computing Subsystem Interfaces. In *Proceedings of the Second International Software Architecture Workshop*.

[Mancoridis and Holt, 1995] Mancoridis, S. and Holt, R. C. (1995). Extending Programming Environments to Support Architectural Design. In *Proceedings of the Seventh International IEEE Workshop on Computer-Aided Software Engineering*, pages 110–119.

[Mancoridis and Holt, 1996] Mancoridis, S. and Holt, R. C. (1996). Recovering the Structure of Software Systems Using Tube Graph Interconnection Clustering. In *Proceedings of the 1996 International Conference on Software Maintenance*.

[Meyer, 1988] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall International, Englewood Cliffs, New Jersey.

[Müller et al., 1992] Müller, H. A., Orgun, M. A., Tilley, S. R., and Uhl, J. S. (1992). Discovering and Reconstructing Subsystem Structures Through Reverse Engineering. Technical Report DCS–201–IR, University of Victoria.

[Naur, 1963] Naur, P. (1963). Revised Report of the Algorithmic Language Algol 60. *Communications of the ACM*, 6(1):1–17.

[Parnas, 1972] Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the Study of Software Architectures. *Software Engineering Notes*, 17(4):40–49.

[Rosenblum and Wolf, 1994] Rosenblum, D. and Wolf, A. L. (1994). Formal Software Engineering. In *Tutorial given at ACM SIGSOFT '94: Symposium on the Foundations of Software Engineering*, New Orleans, Louisiana.

[Shaw, 1989] Shaw, M. (1989). Larger Scale Systems Require Higher-Level Abstractions. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 143–146, Pittsburgh, Pennsylvania.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives of an Emerging Discipline*. Prentice Hall, Englewood Cliffs, New Jersey.

[Silberschatz and Peterson, 1988] Silberschatz, A. and Peterson, J. L. (1988). *Operating System Concepts (Alternate Editon)*. Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

[Stroustrup, 1986] Stroustrup, B. (1986). *The C++ Programming Language*. Addison–Wesley Publishing Company, Reading, Massachusetts.