Using Model-Based Development for Better C Designs

Bruce Powel Douglass, Ph.D.

Bruce.Douglass@us.ibm.com

Twitter: @IronmanBruce

www.bruce-douglass.com

What's wrong with Good Old C?

- Answer:
  Source code is a necessary but insufficient structural model of the system
- Why?
  Because to understand complex systems you need to understand
  - How pieces of different scale and abstraction work together
  - How different aspects (structural, behavioral) of the systems work
- Code is a 1-dimensional, very detailed structural view
- Every other view must be inferred
  - High level structure
  - Dynamic behavior
- With large systems, code-based systems are unmanageable!
  - Expensive to construct
  - Expensive to maintain
  - Expensive to modify
  - Expensive to port
  - Difficult to understand

Models improve

- Visualization
- Understanding
- Communication
- Consistency
- Provability
- Maintainability
- Reusability

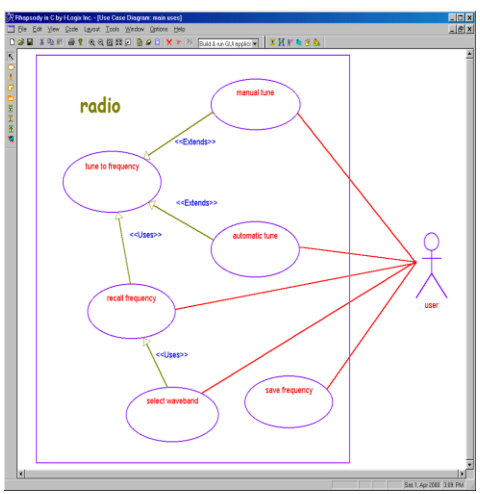
Models

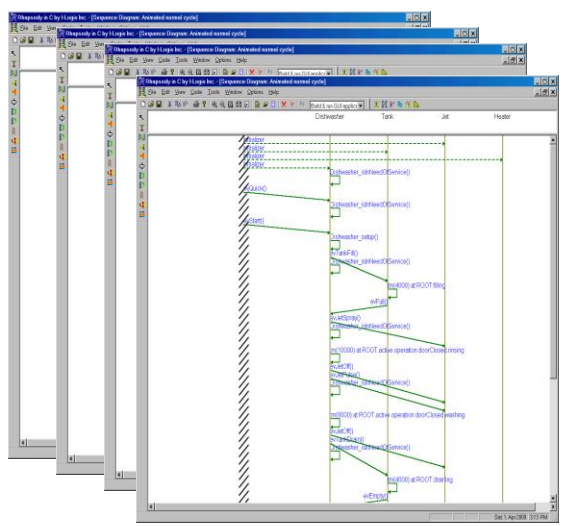
- Graphical models use 2+ dimensions to display structural and behavioral aspects
- Graphical models use abstraction to view the system at different levels of scale from
  - System (very large)
  - Subsystem (large)
  - Component (medium)
  - Class (small)
  - Operation (code) (very small)
- Graphical notations lend themselves to recursive application, allowing any number of levels of abstraction to be used.

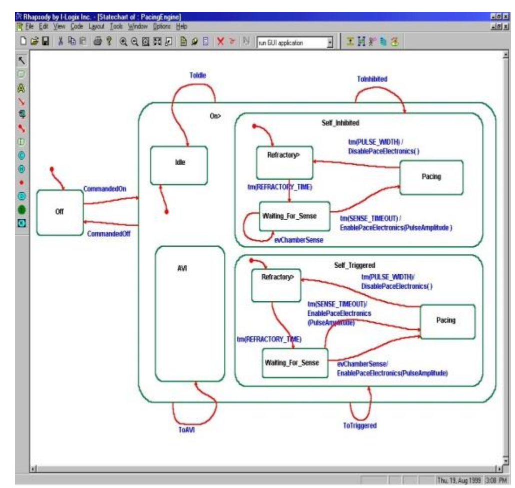
Model-Code Associativity Principle
The code is merely one view of the model.

Visualization

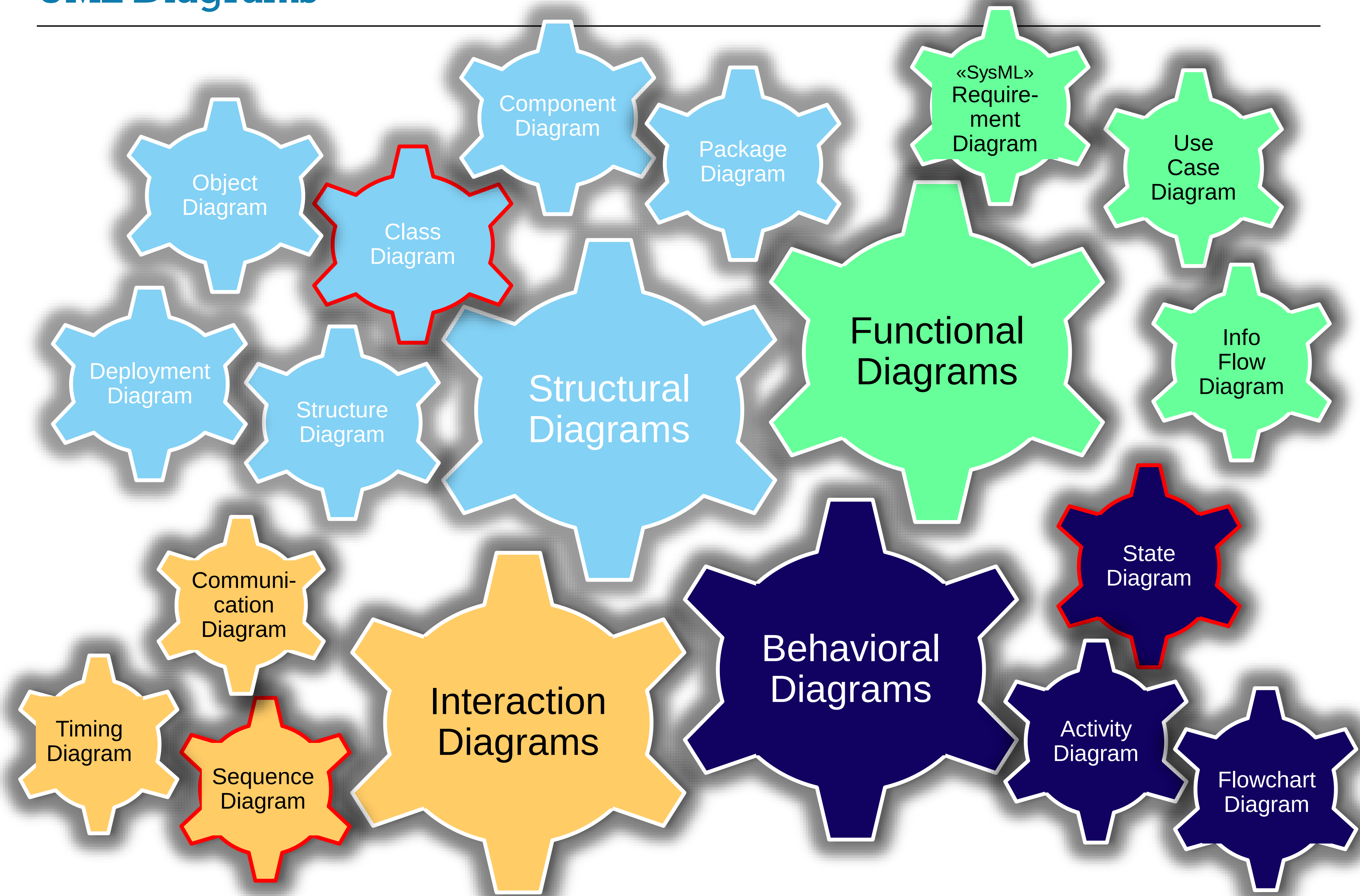
- Visual Models aid
  - Initial construction of the system
  - Ongoing maintenance of the system
  - The testing of the system
  - Introducing new staff to the system
  - Communicating system concepts to others with
    - The appropriate level of abstraction to the concept(s) at hand
    - The appropriate aspects of the system





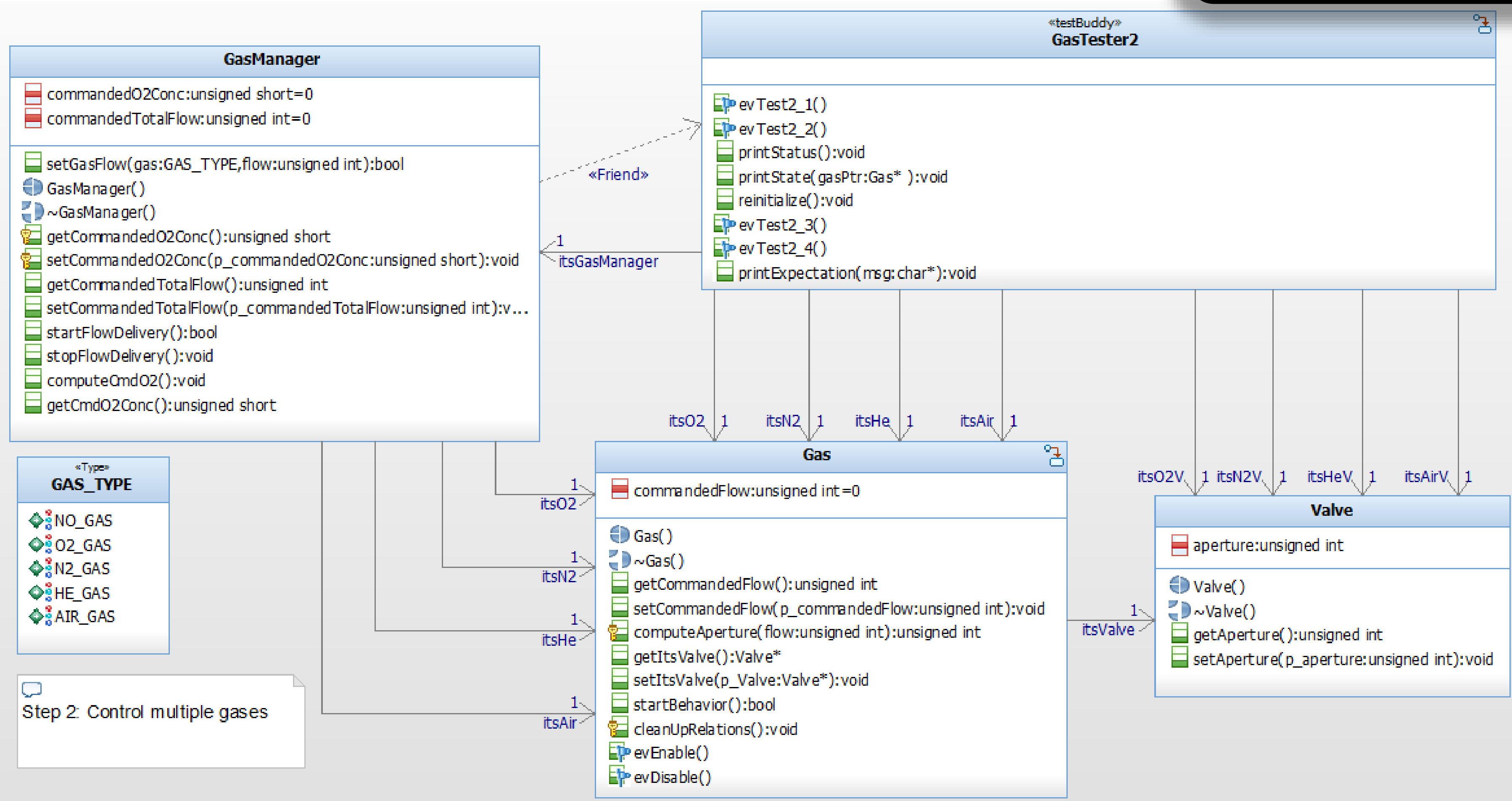


UML Diagrams



Class Diagram

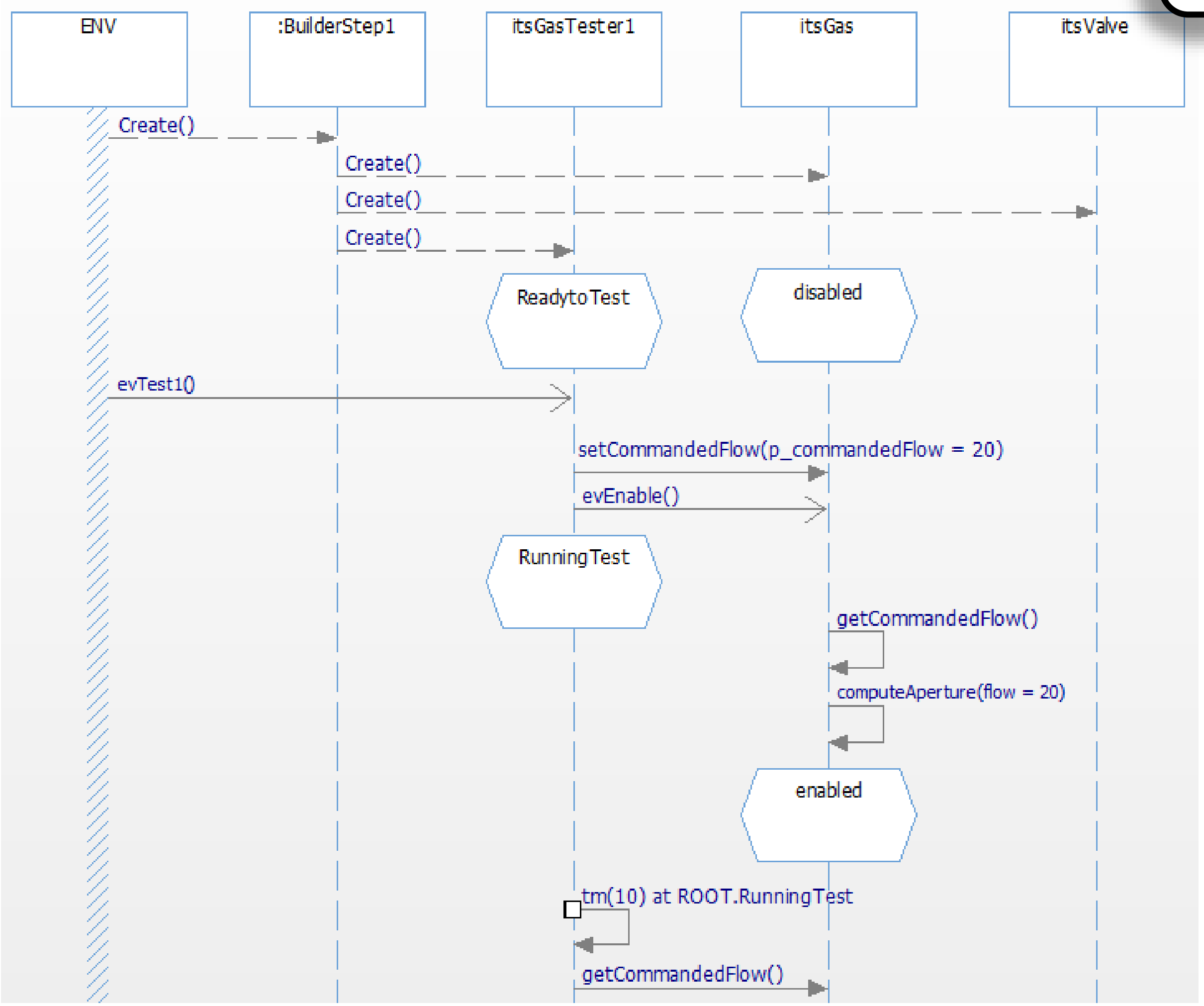
Critical



Class diagrams show structural elements and relations between among.

Sequence Diagram

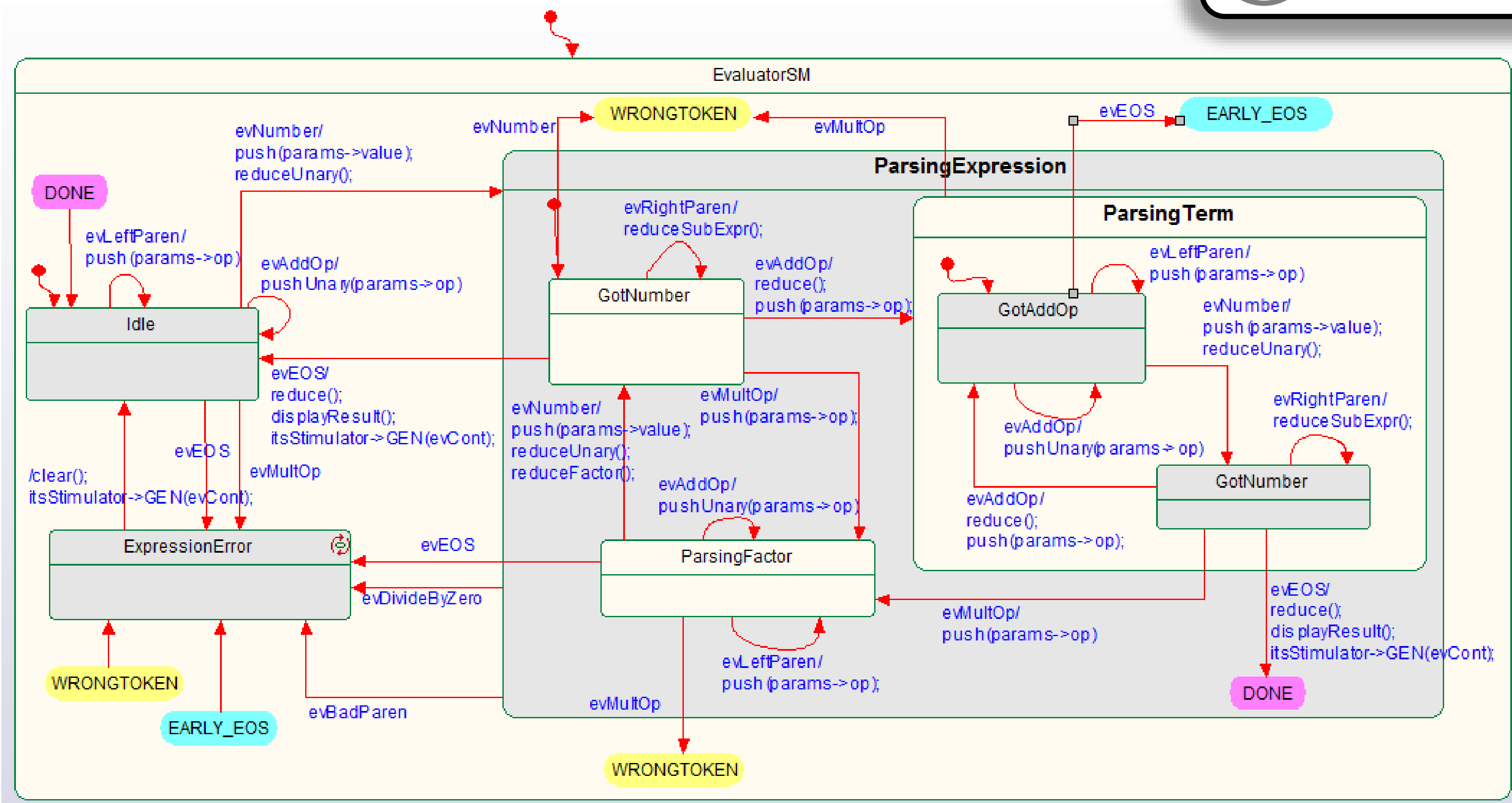
Critical



Sequence Diagrams show how instances communicate over time.

State Machine Diagram

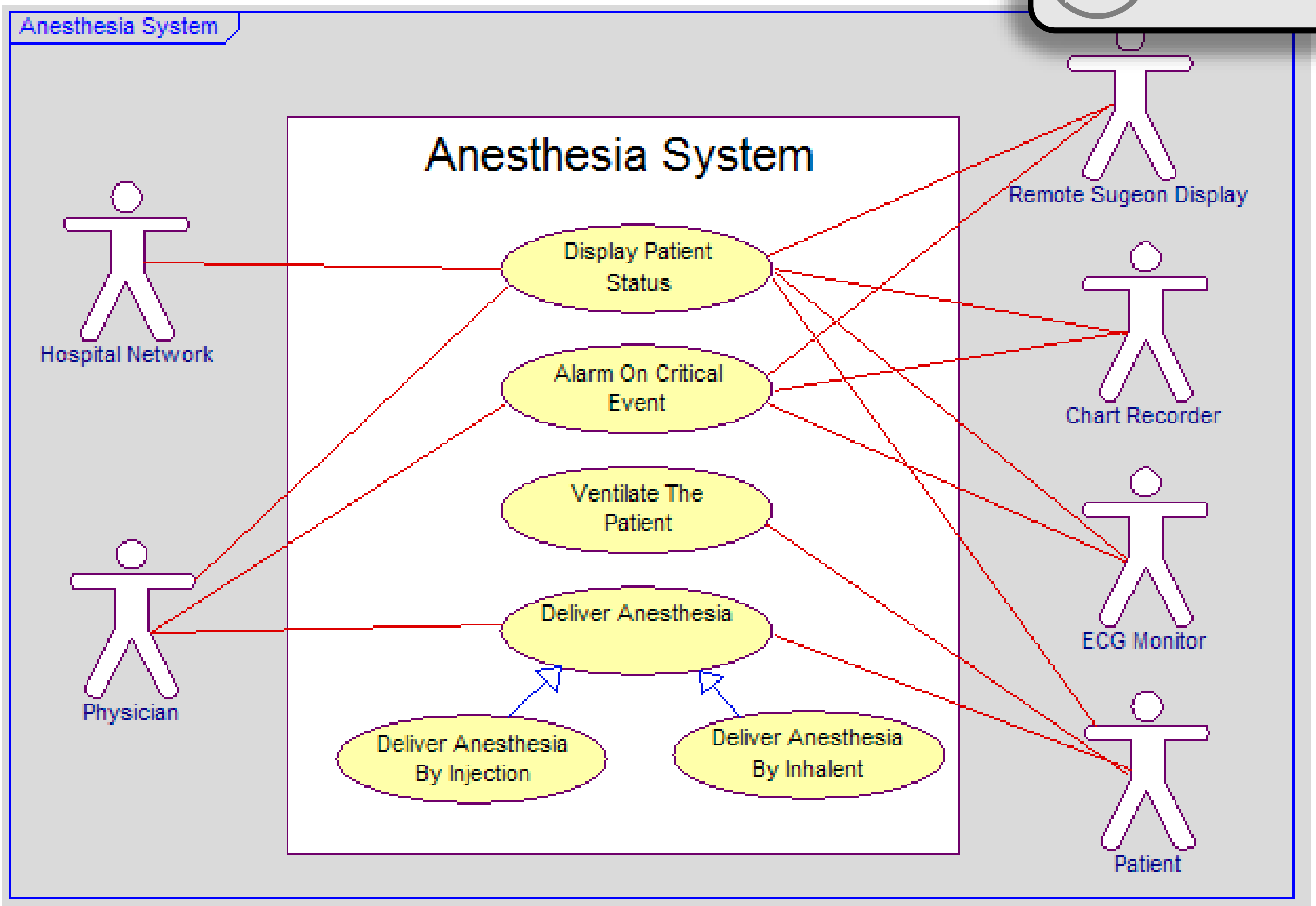
Critical



State machines are used to describe elements whose behavior is state-based and event-driven

Use Case Diagram

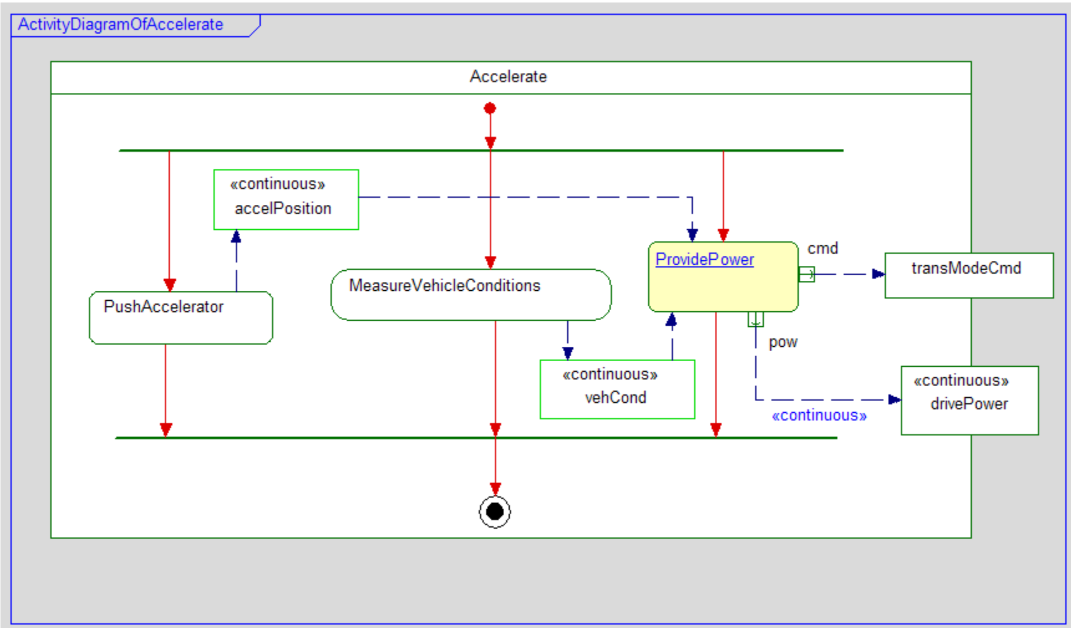
Secondary

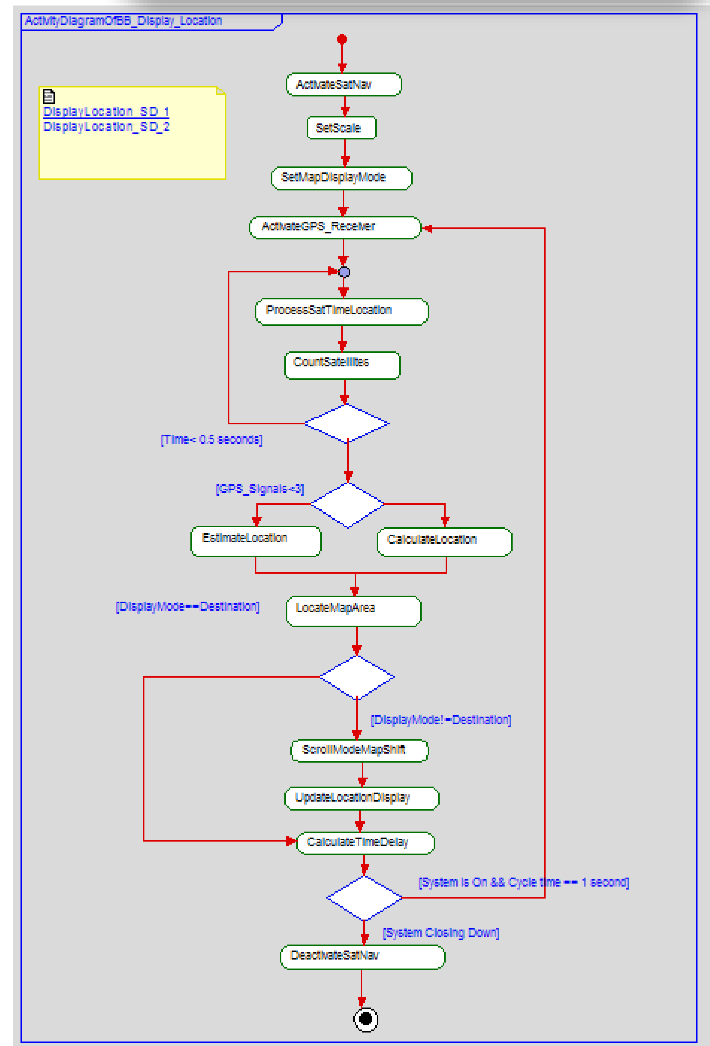


This diagram shows what the system does and who/what interacts with it

Activity Diagram

Secondary





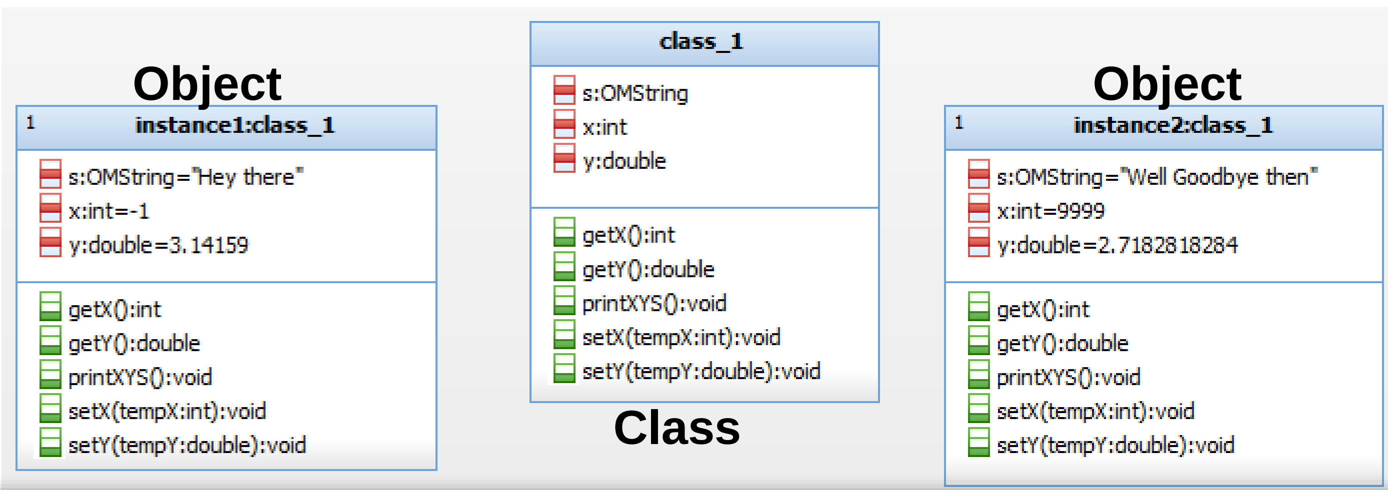
Activity diagrams are used to describe algorithmic behavior for operations, classes or use cases.

UMMI – UML Maturity Model Index

Level	Benefit	Focus	Technologies	Result
5 Optimizing	100%	Agile and Engineering Best Practices	Model-based testing, nanocycle execution, test driven development, continuous integration	Productivity and Quality
4 Executing	70%	Model-based verification	Model execution, code generation, model-based debugging	
3 Behavioral Modeling	30%	State and algorithmic modeling	State, sequence and activity diagrams	
2 Structural Modeling	15%	Class and block modeling of structure	Class and block diagrams	
1 Visualization	5%	Visualizing code structures	Reverse engineering	
0 Code Based Development	0%	Manual, time intensive heroic development		

Classes and Objects

- A class is a design-time specification that defines the structure and behavior for a set of objects to be created at run-time.
  - Specifies behavior implementation (methods)
  - Specifies data (attributes)
  - Specifies state (optional)
- An object is a run-time entity that occupies memory at some specific point in time
  - Instance of a class
    - That means it is a new set of data defined by the class and owned by the instance

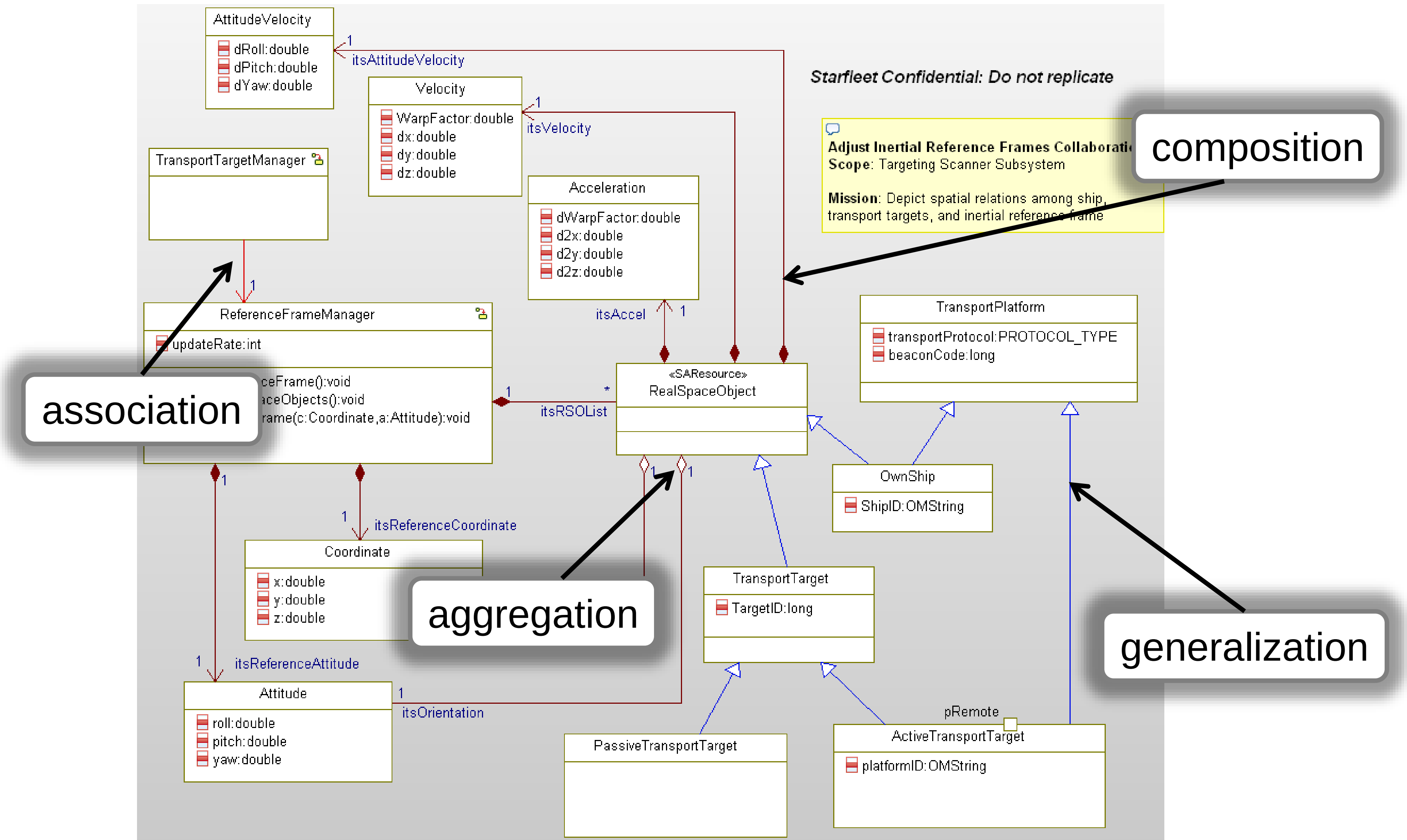


Hmmm. A class has operations (functions and event receptions), data, and types. Doesn't that kind of sound like a standard C file?

Relations

- Relations allow objects to communicate at run-time or to share metadata at design time
- Class may use the facilities of other classes with an association
  - Note: Objects are connected via links. Links are instances of associations
    - That is: an association defines a pointer from the source to destination class. The link is the actual pointer value within an instance in the running system
- There are two specialized forms of association:
  - Classes may strongly aggregate others (as parts defined by other classes) via composition
- Classes may derive attributes and behaviors from other classes with a generalization
- Classes may depend on others via a dependency
  - Classes may contain others with an aggregation

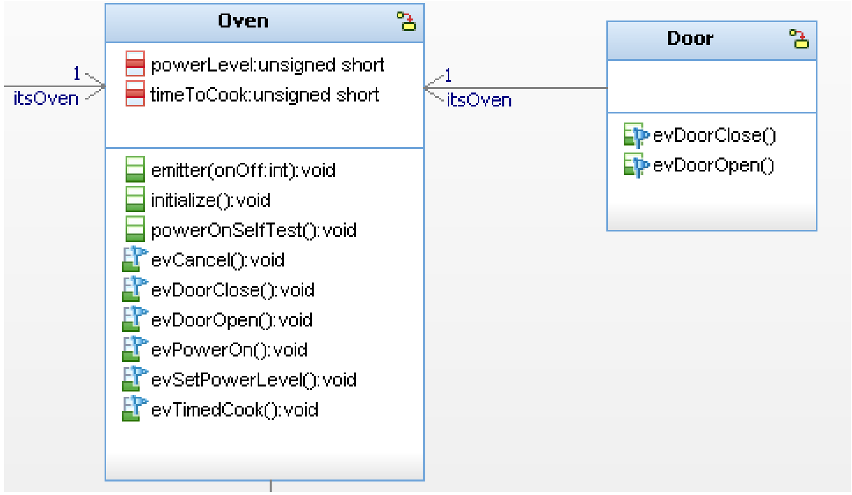
Relations



AttitudeVelocity
dRoll:double
dPitch:double
dYaw:double
1
itsAttitudeVelocity
Velocity
WarpFactor:double
dx:double
dy:double
dz:double
1
itsVelocity
Starfleet Confidential: Do not replicate
Adjust Inertial Reference Frames Collaborati
Scope: Targeting Scanner Subsystem
Mission: Depict spatial relations among ship, transport targets, and inertial reference frame
composition
TransportTargetManager
Acceleration
dWarpFactor:double
d2x:double
d2y:double
d2z:double
itsAccel
1
ReferenceFrameManager
updateRate:int
ceFrame():void
aceObjects():void
rame(c:Coordinate,a:Attitude):void
association
«SAResource»
RealSpaceObject
itsRSOList
TransportPlatform
transportProtocol:PROTOCOL_TYPE
beaconCode:long
OwnShip
ShipID:OMString
itsReferenceCoordinate
Coordinate
x:double
y:double
z:double
aggregation
TransportTarget
TargetID:long
generalization
itsReferenceAttitude
Attitude
roll:double
pitch:double
yaw:double
itsOrientation
pRemote
PassiveTransportTarget
ActiveTransportTarget
platformID:OMString

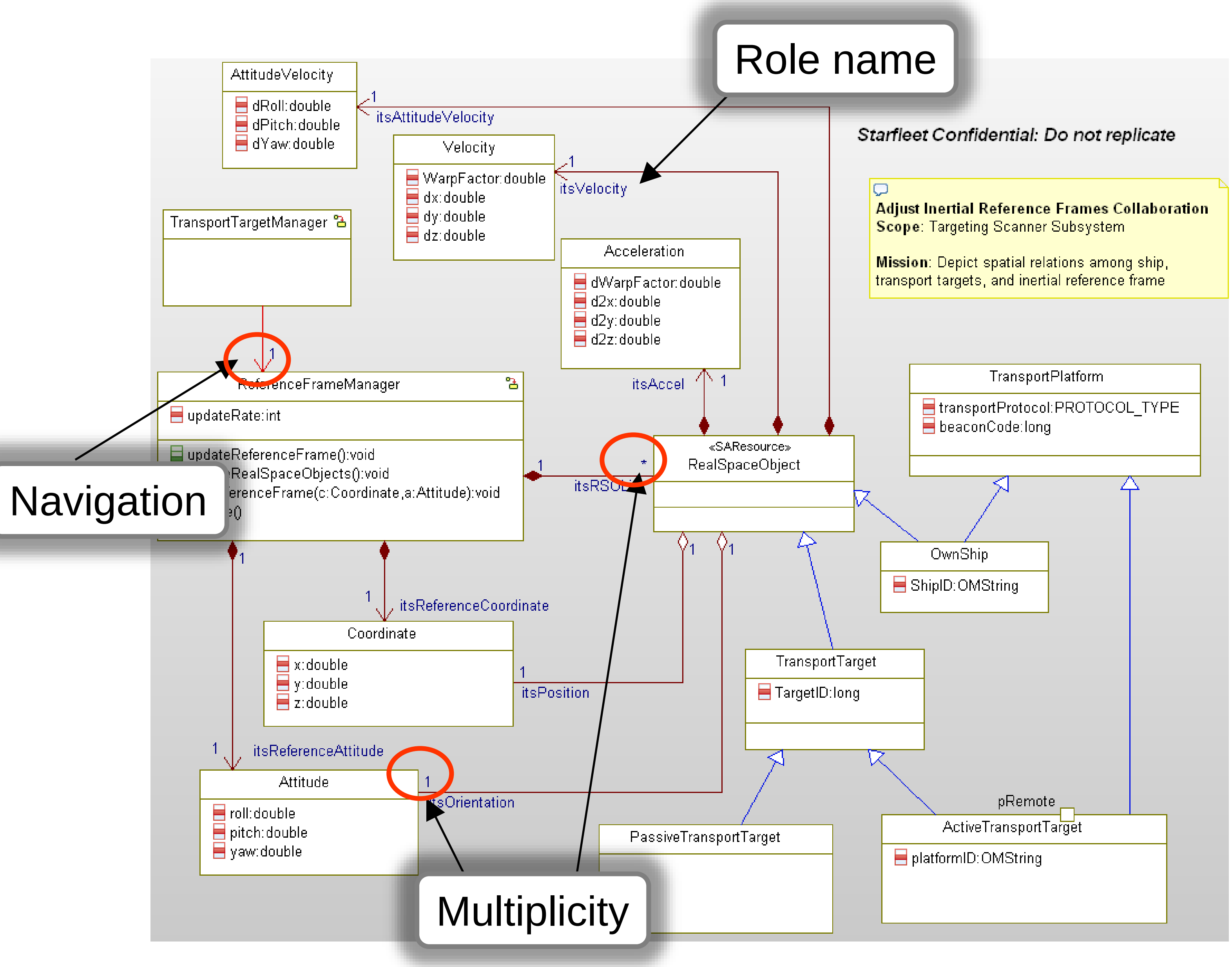
Associations – the most important relation

- Associations may have labels
  - This is the “name” of the association
  - Labels are little used



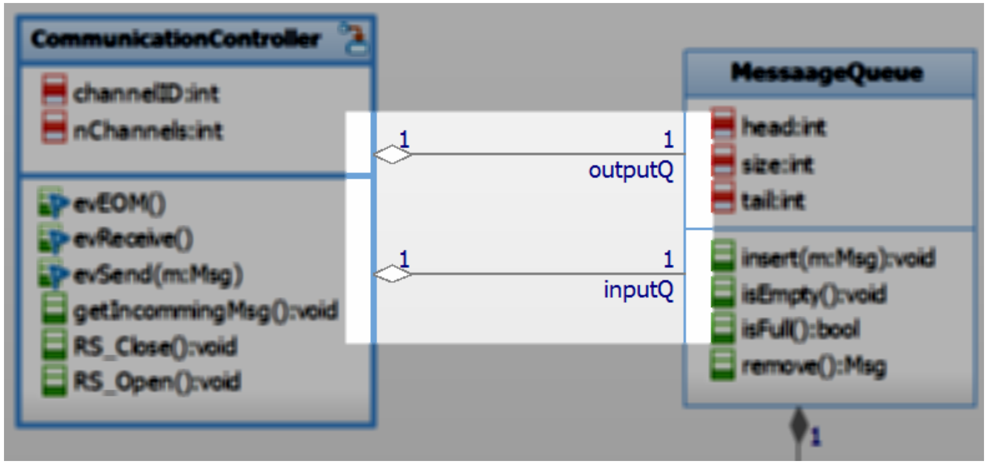
- Associations may have role names
  - Identifies the role of the object in the association
  - Implementation hint: this is usually the name of the pointer realizing the relation
- Associations may indicate multiplicity
  - Identifies the number of instances of the class that participate in the association
  - Implementation hint: >1 multiplicities can be done with arrays or container classes (e.g. linked lists)
- Associations may indicate navigation with an open arrowhead
  - Unadorned associations are assumed to be bi-directional
  - Most associations are unidirectional

Multiplicity / Navigation

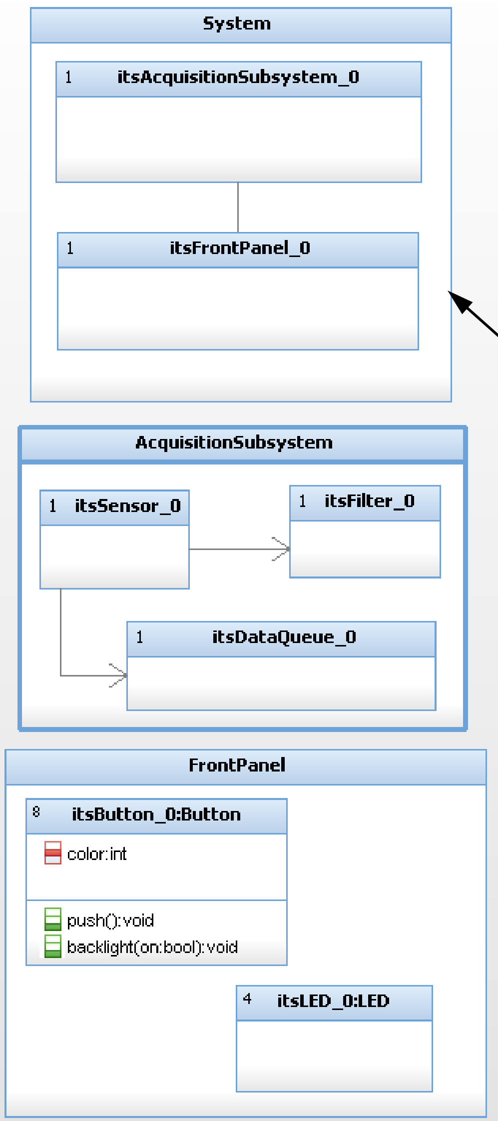


Aggregation (remember – it's an association!)

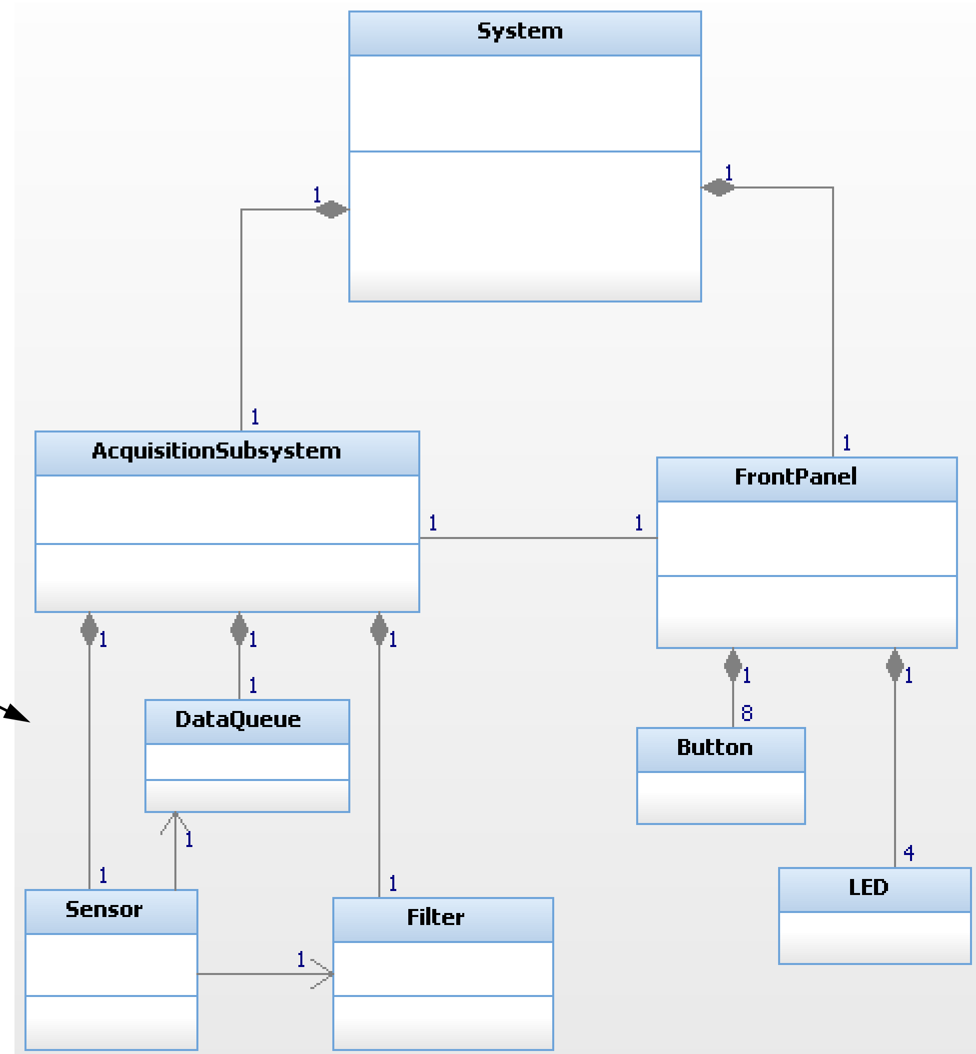
- Is a kind of association
- Indicated by a hollow diamond
- "Whole-part" relationship
  - Denotes one object logically or physically contains another
- "Weaker" form of aggregation. Nothing is implied about
  - Navigation
  - Ownership
  - Lifetimes of participating objects



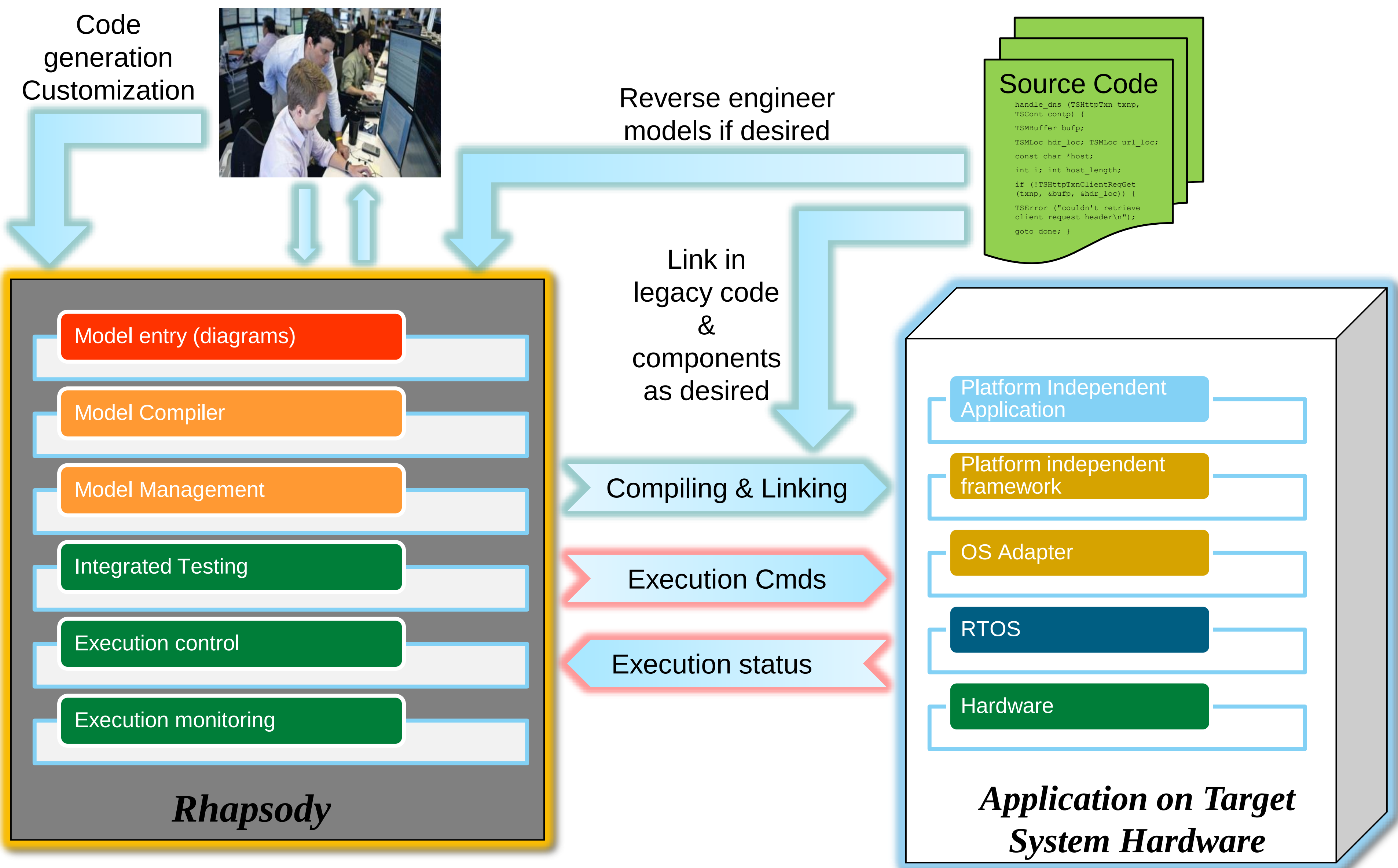
Composition – associations with responsibility



Two ways to show composition



UML Software Development Environment

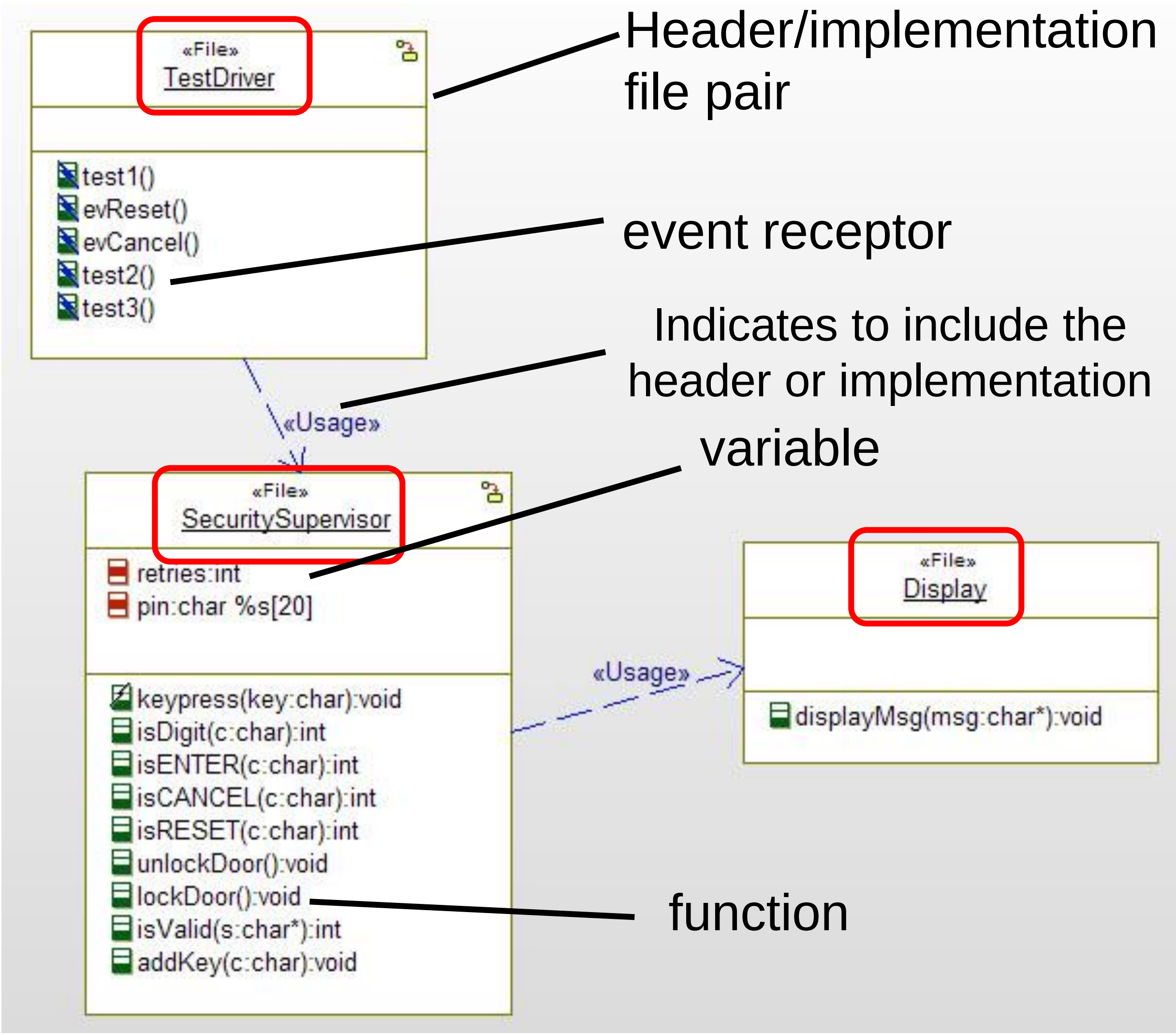


Three primary approaches to use UML with C

- Functional design (FD)
  - Based on the notion of Files
  - Files contain
    - Data
    - Functions
    - Data types
- Object-based design (OBD)
  - Support objects by creating structs
  - Bind functions to structs using naming conventions (“name mangling”)
  - No inheritance or generalization
  - Use a me pointer to identify which data instance to the functions
- Object-oriented design (OOD)
  - Support generalization and inheritance through creating virtual function tables within structs
  - Bind functions to structs using function pointers
  - Use a me pointer to identify which data instance to the functions
- In all cases,
  - Generate both .c and .h files
  - State machines have class or file scope

Classes represented as Files in C

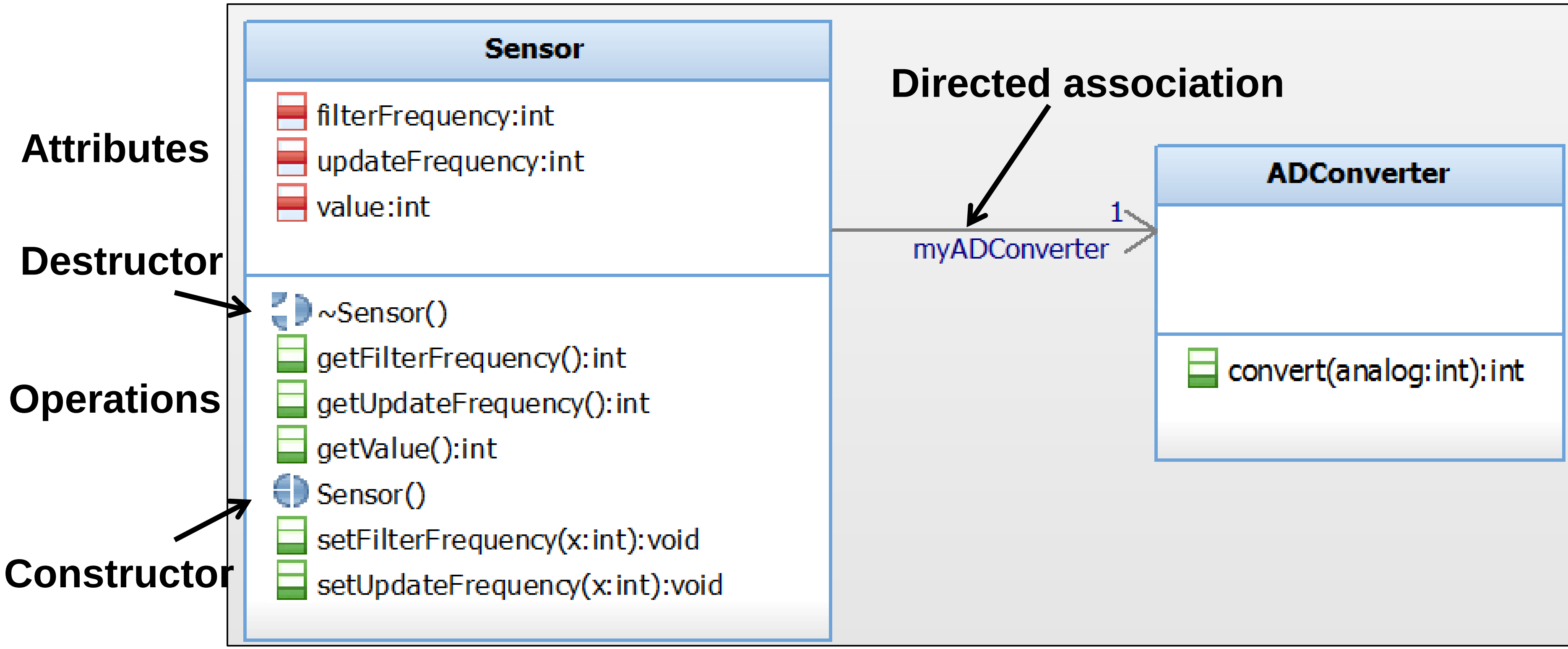
- A file (*.h and *.c) lumps together
  - Variables
  - Event types
  - Functions (including state machine implementations)
  - Types and typedefs
  - Preprocessor declarations
  - «File» shows that the “class” is representing the contents of a header/implementation file pair
  - «Usage» indicates include the header file



Classes as Files in C

- “Traditional” C development
- Heavy use of singletons (single instances)
  - Little use of multiple instances
- Associations implemented via pointers or references
- No structs needed
- No generalization used

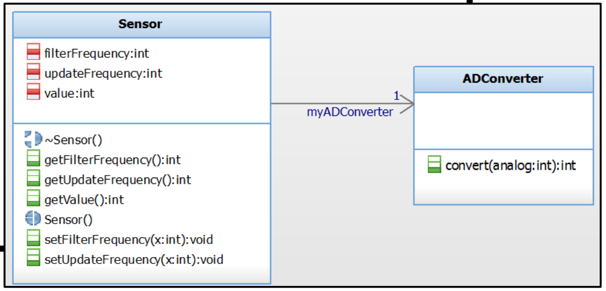
Let's use an example for OBD and OOD



C Object Based Design (header file)

- The **`me`** pointer points to instance data (supports multiple instances of class)

```c
#ifndef Sensor_H
#define Sensor_H
#include "ADConverter.h"

/* class Sensor */
typedef struct Sensor Sensor;
struct Sensor {
    int filterFrequency;
    int updateFrequency;
    int value;
    ADConverter* myADConvert; /* association implemented as ptr */
};
int Sensor_getFilterFrequency(const Sensor* const me);

void Sensor_setFilterFrequency(Sensor* const me, int p_filterFrequency);

int Sensor_getUpdateFrequency(const Sensor* const me);

void Sensor_setUpdateFrequency(Sensor* const me, int p_updateFrequency);

int Sensor_getValue(const Sensor* const me);

Sensor * Sensor_Create(void);

void Sensor_Destroy(Sensor* const me);
#endif
```



C Object Based Design (implementation file)

```c
#include "Sensor.h"

int Sensor_getFilterFrequency(const Sensor* const me) {
    return me->filterFrequency;
}

void Sensor_setFilterFrequency(Sensor* const me, int
p_filterFrequency) {
    me->filterFrequency = p_filterFrequency;
}

int Sensor_getUpdateFrequency(const Sensor* const me) {
    return me->updateFrequency;
}

void Sensor_setUpdateFrequency(Sensor* con
p_updateFrequency) {
    me->updateFrequency = p_updateFrequenc
}

int Sensor_getValue(const Sensor* const me
    return me->value;
}
```

```c
         /* Constructor and destructor */
Sensor * Sensor_Create(void) {
    Sensor* me = (Sensor *) malloc(sizeof(Sensor));
    if(me!=NULL)
        {
            Sensor_Init(me);
        }
    return me;
}

void Sensor_Destroy(Sensor* const me) {
    if(me!=NULL)
        {
            Sensor_Cleanup(me);
        }
    free(me);
}
```

C Object Based Design

- Use of structs to represent classes
- Supports multiple instances
  - Adds a me pointer to the struct instance as the first parameter of all class functions to identify which object's data should be acted on
- Class functions are prepended with the class name
  - A class **Sensor** with a function **acquire()** would internally name the function **Sensor_acquire()**
- No use of generalization

C Object Oriented Design (header file)

- The function pointers support polymorphism and virtual functions

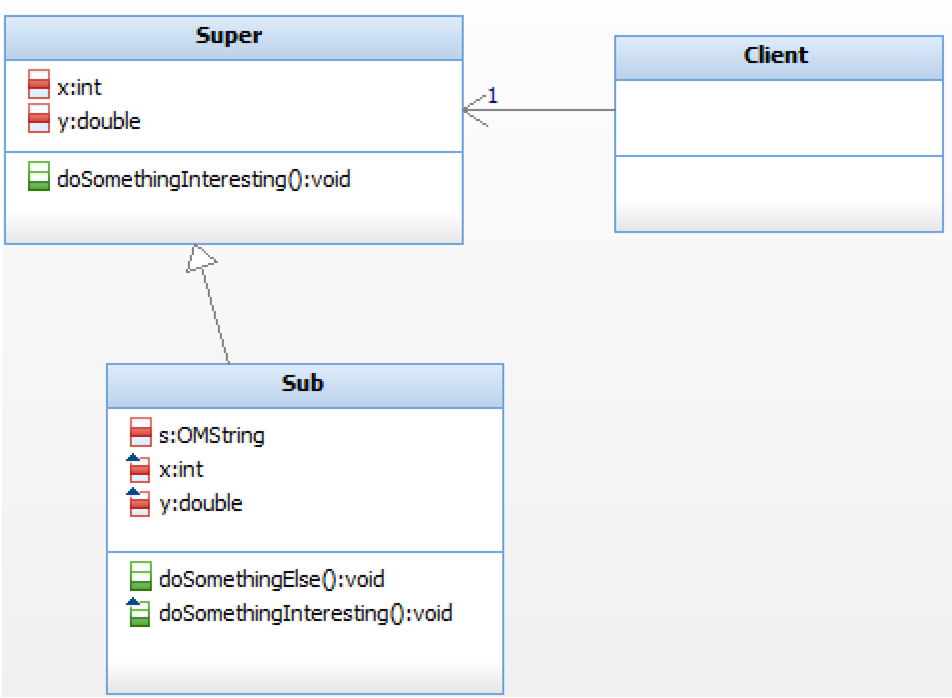
```
#ifndef Sensor_H
#define Sensor_H
#include "ADConverter.h"

    /* function pointers */
typedef int (*f0ptrInt)(void*);
typedef void (*f1ptrVoid)(void*,int);

/* class Sensor */
typedef struct Sensor Sensor;
struct Sensor {
    int filterFrequency;
    int updateFrequency;
    int value;
    ADConverter* myADConvert; /* association implemented as ptr */
    f0ptrInt getFilterFreq;  /* ptr to the function w only me ptr argument */
    f1ptrVoid setFilterFreq; /* ptr to function with me ptr and int args */
};

int getFilterFrequency(const Sensor* const me);
Void setFilterFrequency(const Sensor* const me, int ff);

Sensor * Sensor_Create(void); /* creates struct and calls init */
Void Sensor_Init(Sensor* const me); /* intializes vars incl. function ptrs */

void Sensor_Destroy(Sensor* const me);
#endif
```

```
/* initialize function ptrs in constructor */
void Sensor_Init(Sensor* const me) {
    me->getFilterFreq = subGetFilterFrequency;
    me->setFilterFreq = subSetFilterFrequency;
}
```

C Object Oriented Design

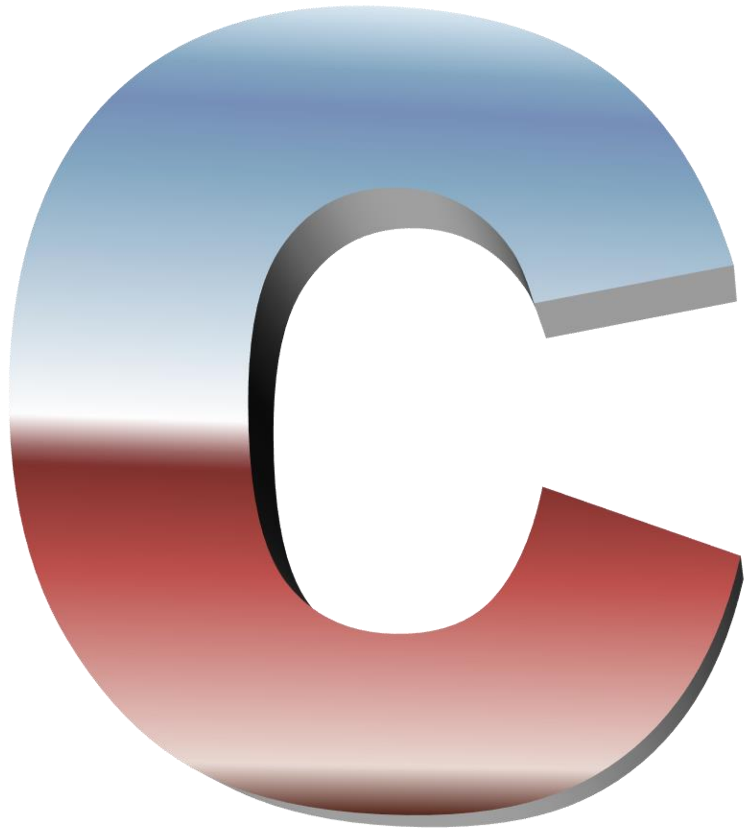
- Use of structs to represent classes
- Supports multiple instances
  - Adds a me pointer to the struct instance as the first parameter of all class functions to identify which object's data should be acted on
- Supports generalization
  - Subclasses include the base class structure and extend it
  - Polymorphism is supported by having function pointer to functions of interest
    - Requires double dereferencing but it means that at run-time calling a function can refer to the subclasses function because the function pointer points to the replaced function

Generalization and Polymorphism

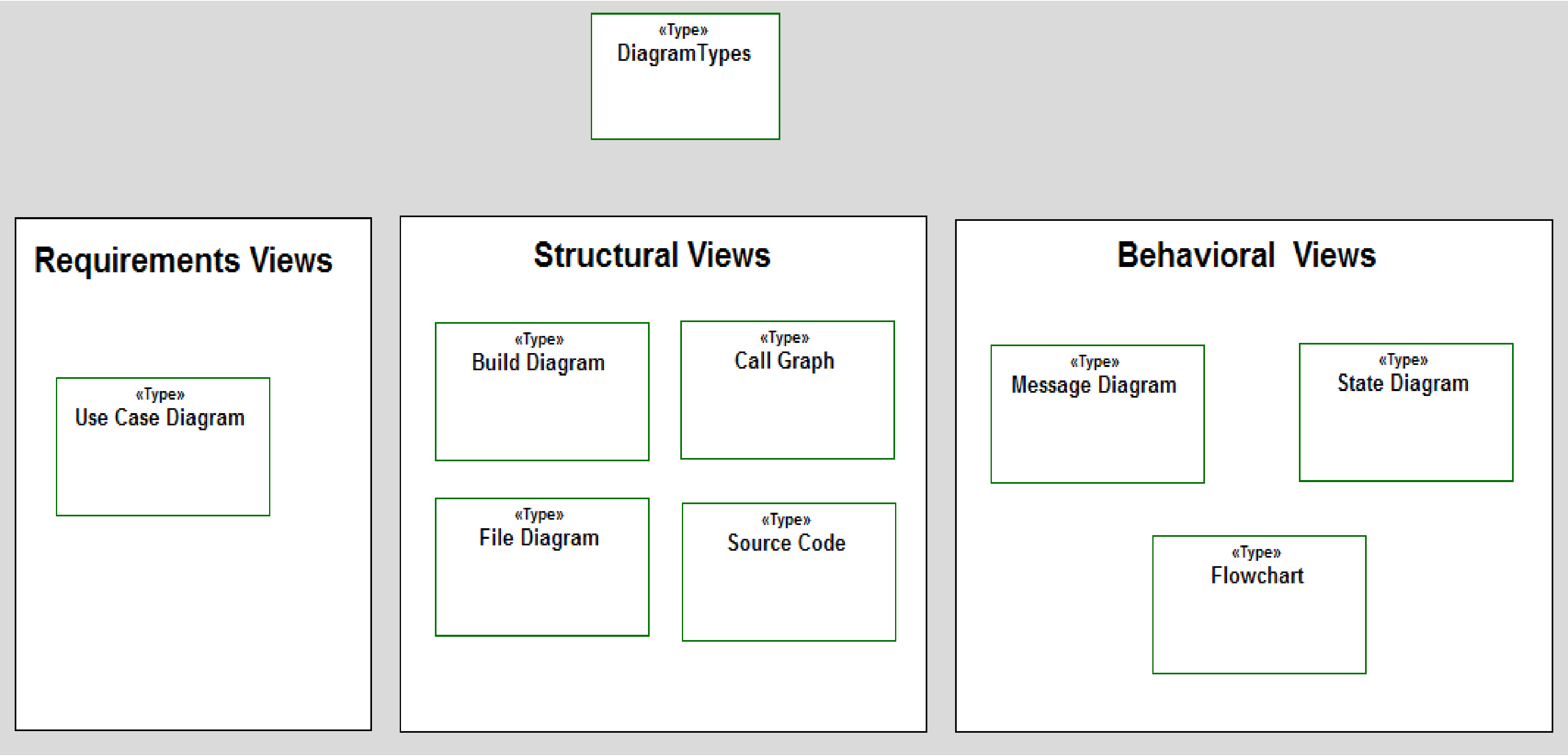


- To the left, “Sub” is a specialized kind of “Super’
- Sub inherits all the data and behavior of Super
- Sub may specialize or extend Super by
  - Adding new data elements
  - Adding new functions
  - Redefining existing functions
  - So if Client calls doSomethingInteresting() but is actually pointing to an instance of Sub, then the Sub implementation is invoked.
  - This can be done in C by invoking the function by referencing a function point. When you create the instance of Sub, point to the (internally named Sub_doSomethingInteresting()) function

Graphical C

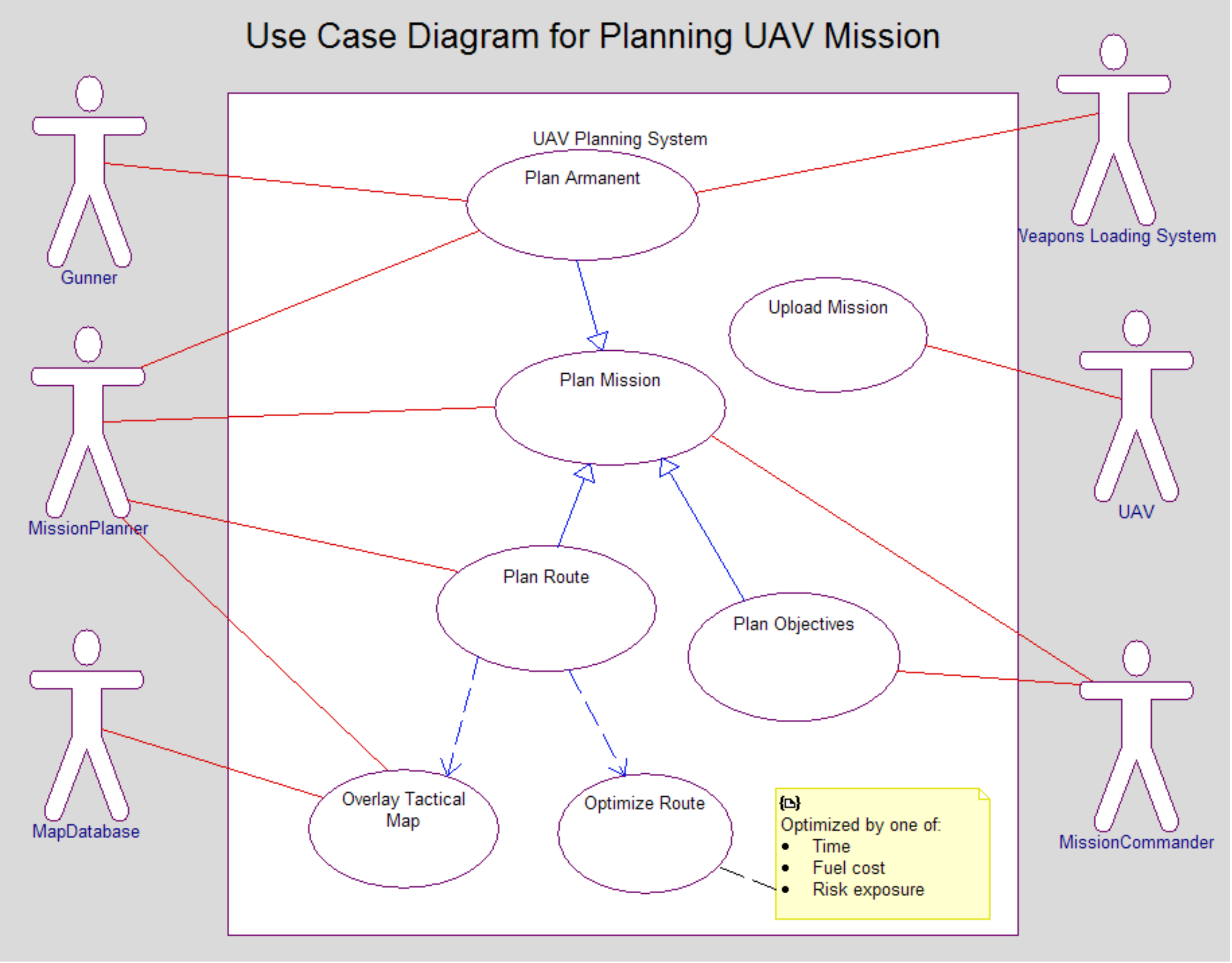


Graphical C Diagrams



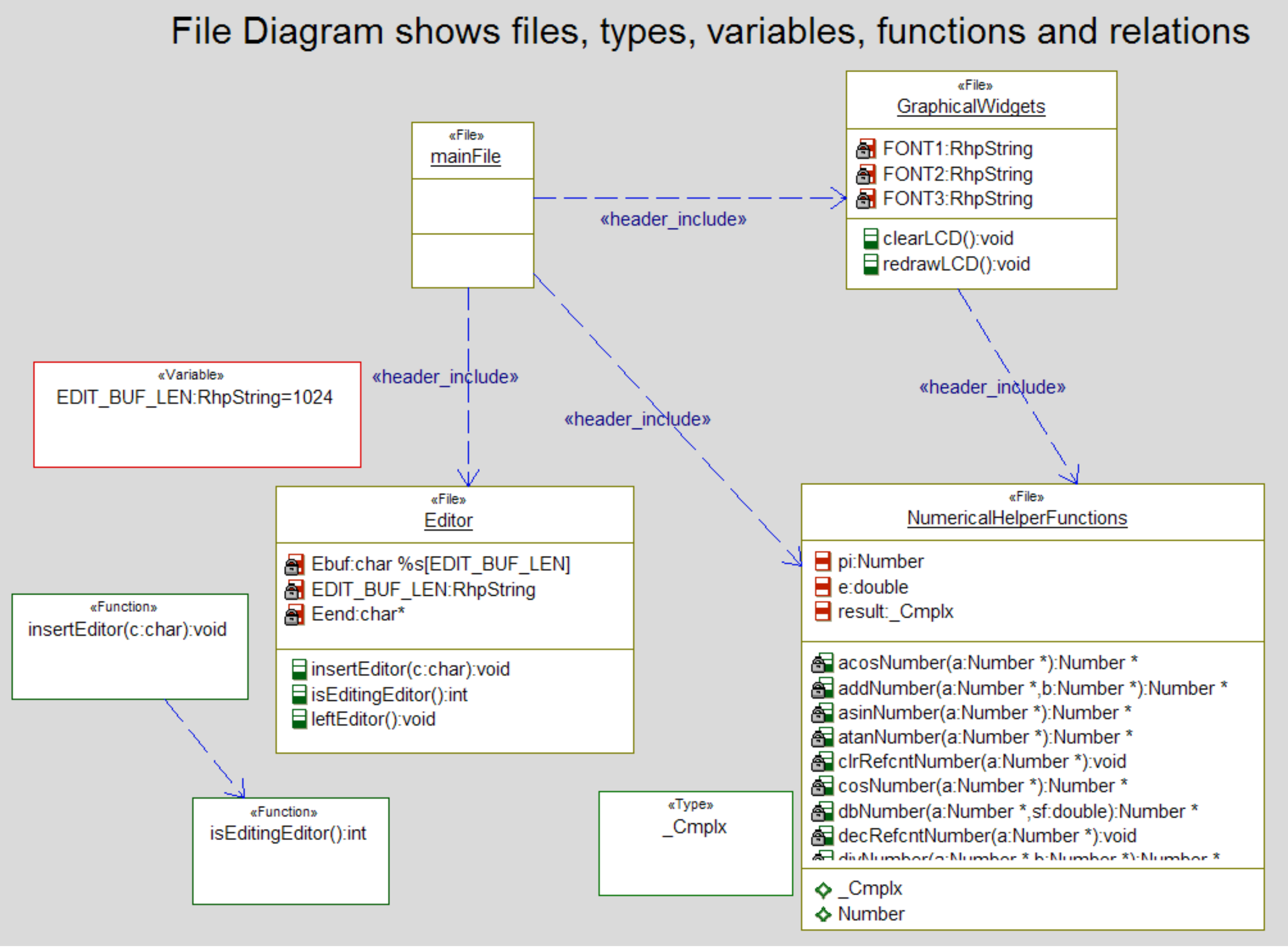
Use Case Diagram

- Used to cluster requirements into “system uses”
- Contains
  - Use Cases
  - Actors
  - Requirements
  - Constraints
  - Comments
  - Relations
    - Association
    - Dependency
    - Generalization



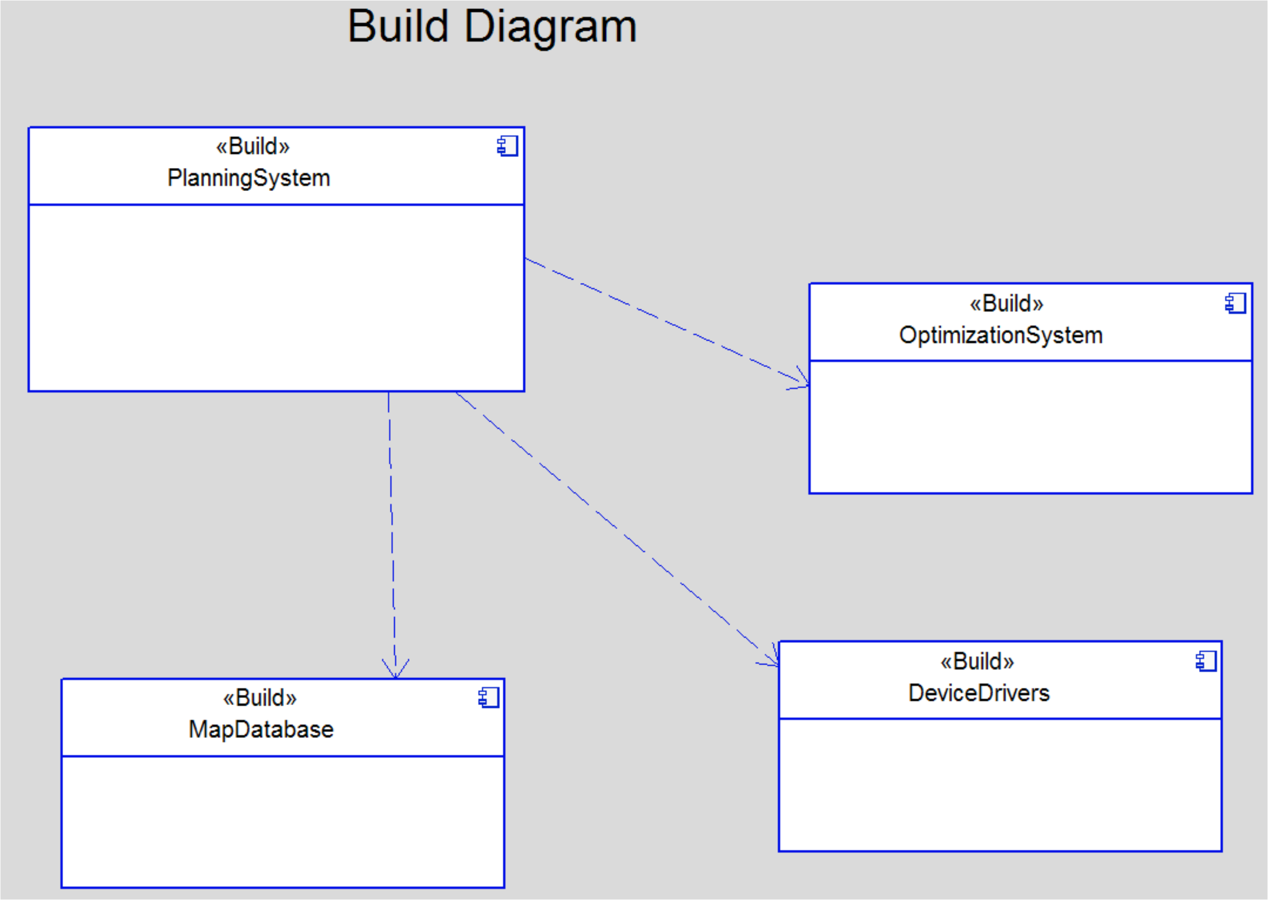
File Diagram

- C programs are typically composed of Files containing
  - Variables
  - Functions
  - Types
  - Includes
    - Header
    - Body



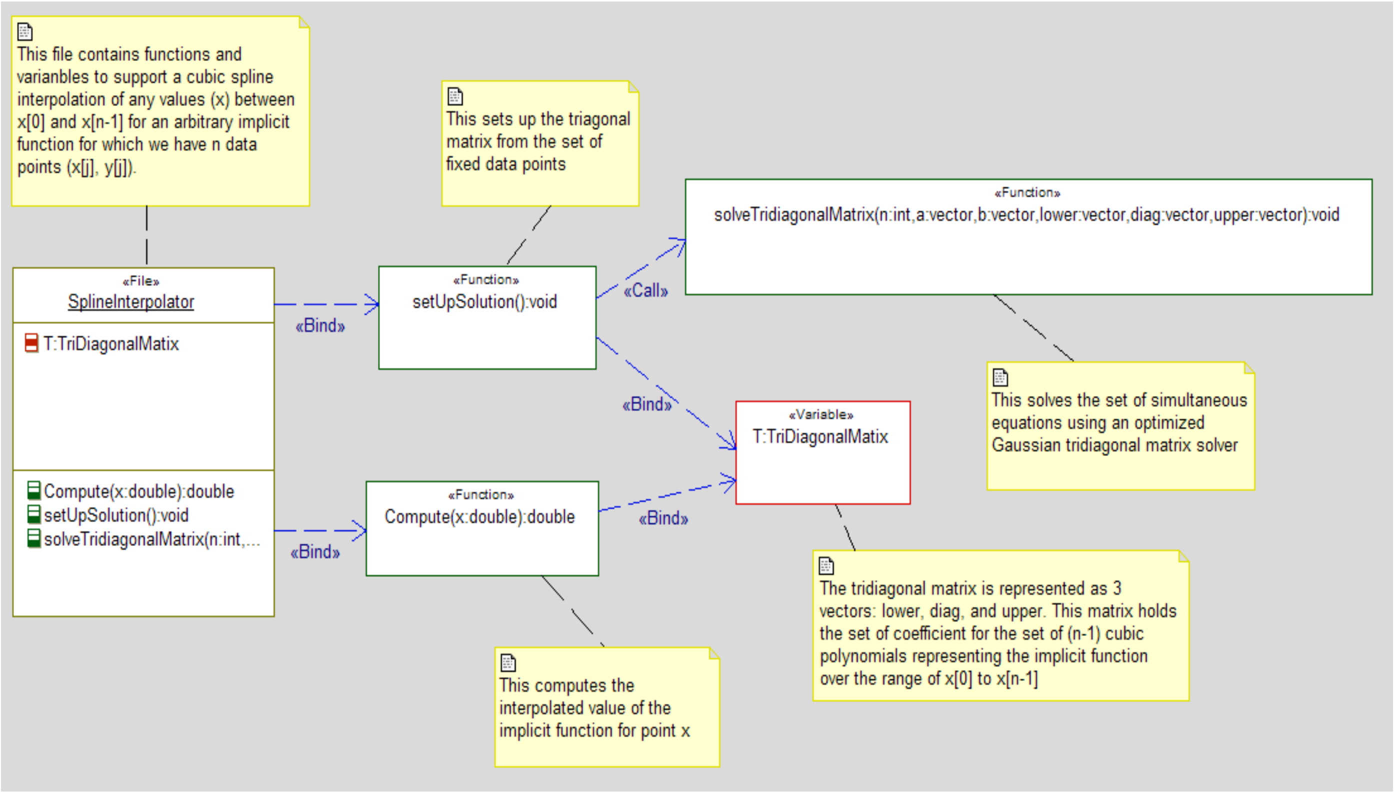
Build Diagram

- Shows the components required to construct the system
- These may be
  - Executables
  - Libraries
  - External Source code



Call Graph

- Shows
  - The calls and their sequence among set of functions
  - The binding of variables and functions
- Note
  - Can only show calls (synchronous function invocations) not asynchronous event receptions



Message Diagram

- Message diagram shows an exemplar (aka scenario) between a set of files, depicting
  - The files as vertical "lifelines"
  - Messages as arrowed lines, which may be either
    - Synchronous (i.e. function calls)
    - Asynchronous (i.e. queued events)
  - Annotations, such as quality of service constraints

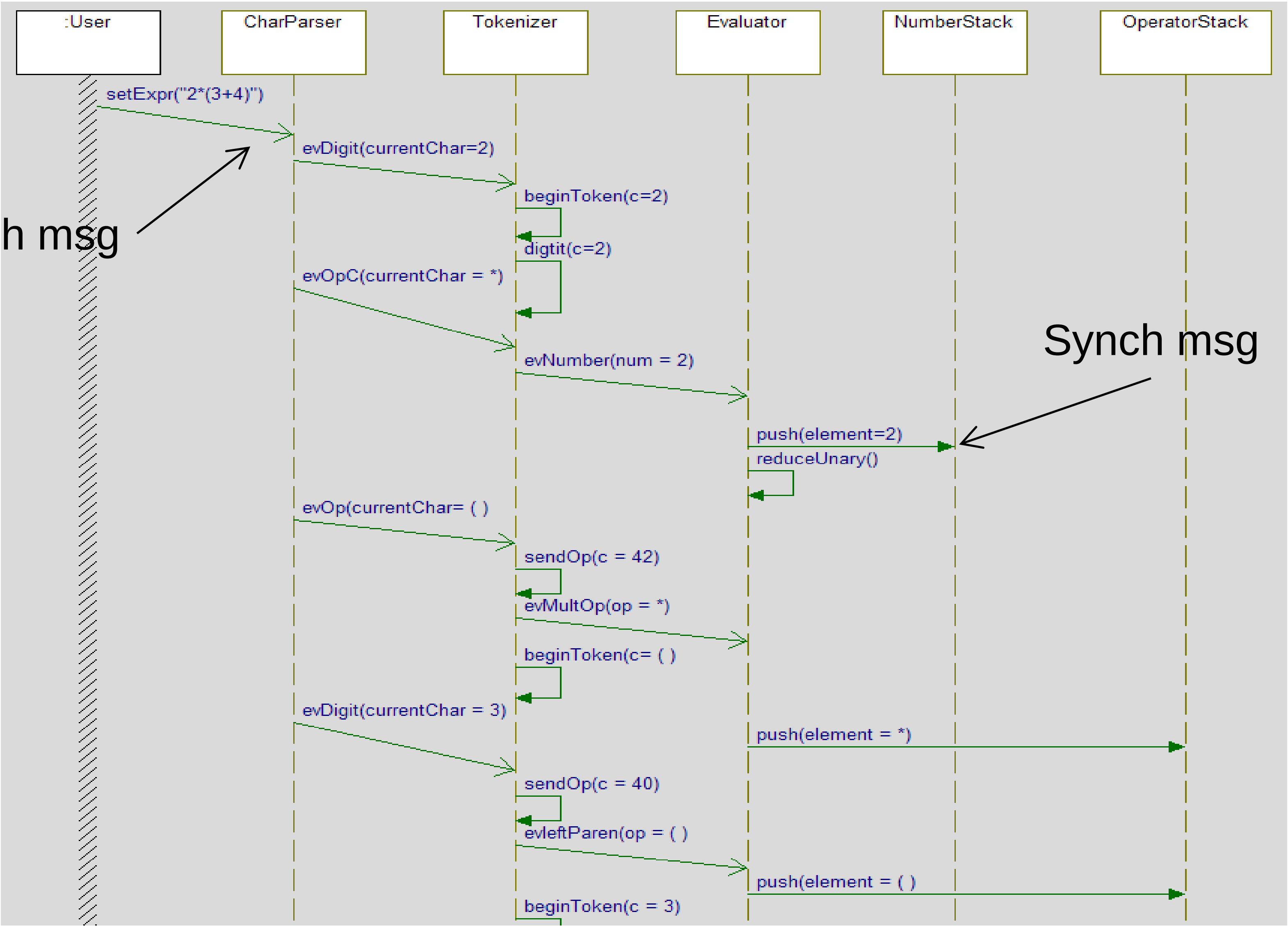
- A system normally has many message diagrams depicting
  - Different messages
  - Different sequences
  - Both

- A message diagram relates to either a flow chart or statechart by showing a singular path through that formal specification

Message Diagram

Asynch msg

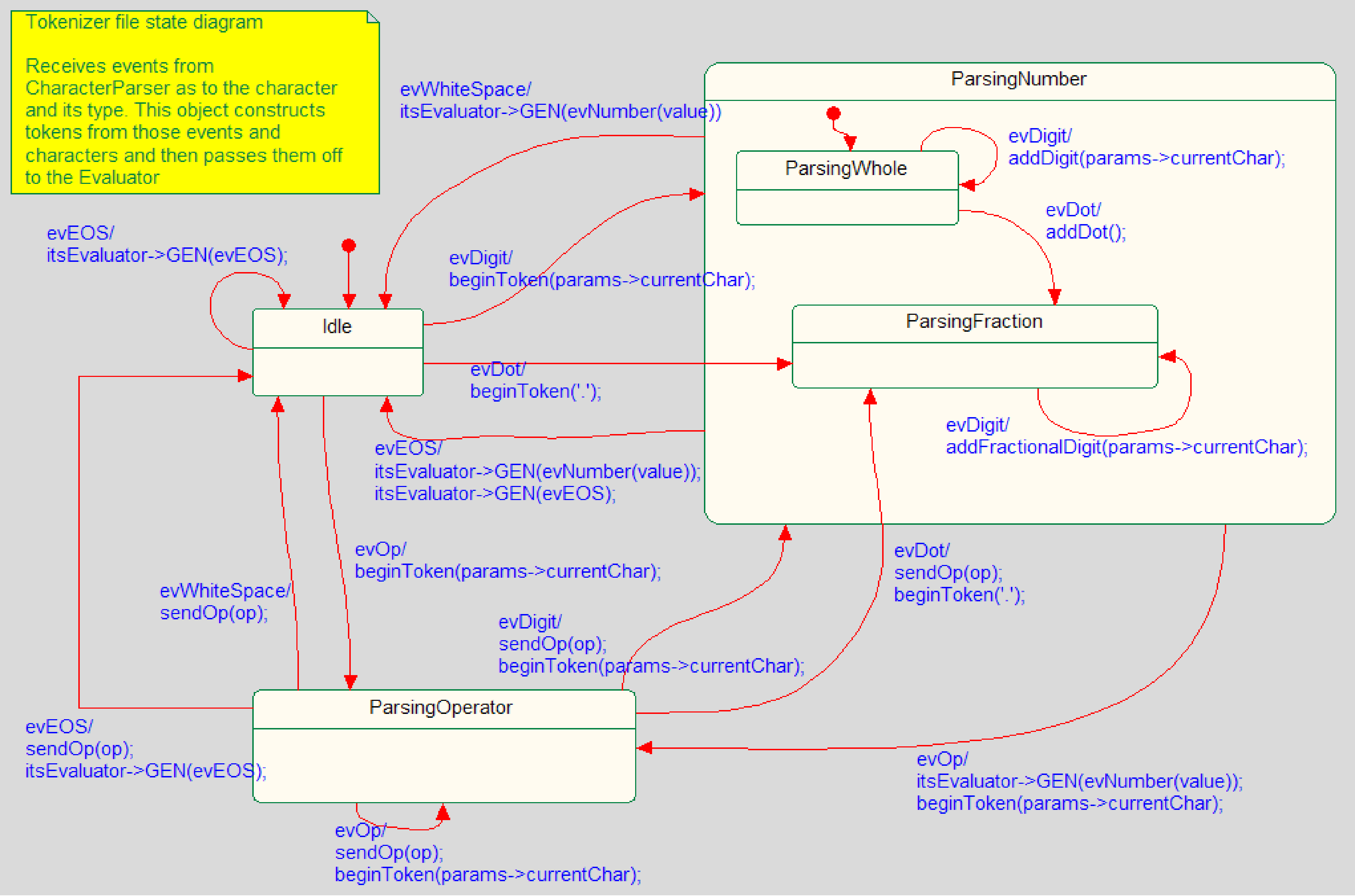
Synch msg



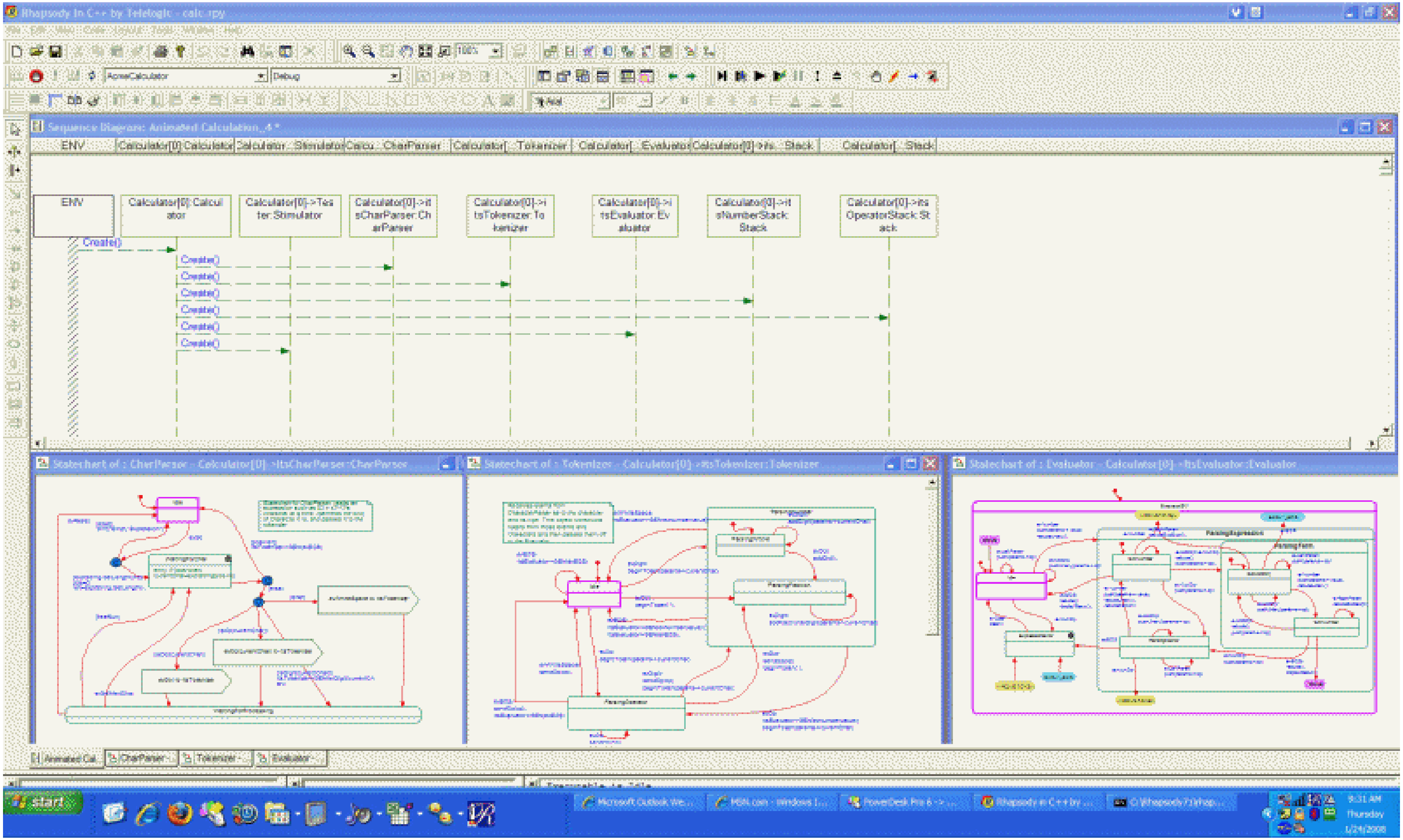
State Diagram

- UML state diagrams are based on Harel Statecharts.
- They depict the state behavior of elements (normally files) and control the sequencing of functional invocations and primitive operations that take place in response to events
- Events may be
  - Synchronous (“triggered functions”)
  - Asynchronous event receptions
  - Timeouts
- States may be nested within states
  - On the same diagram
  - On “nested diagrams”
- State may be
  - “OR-states”
    - Element may be in one state or another
  - “AND-states”
    - Elements may be in multiple states simultaneously

State Diagram

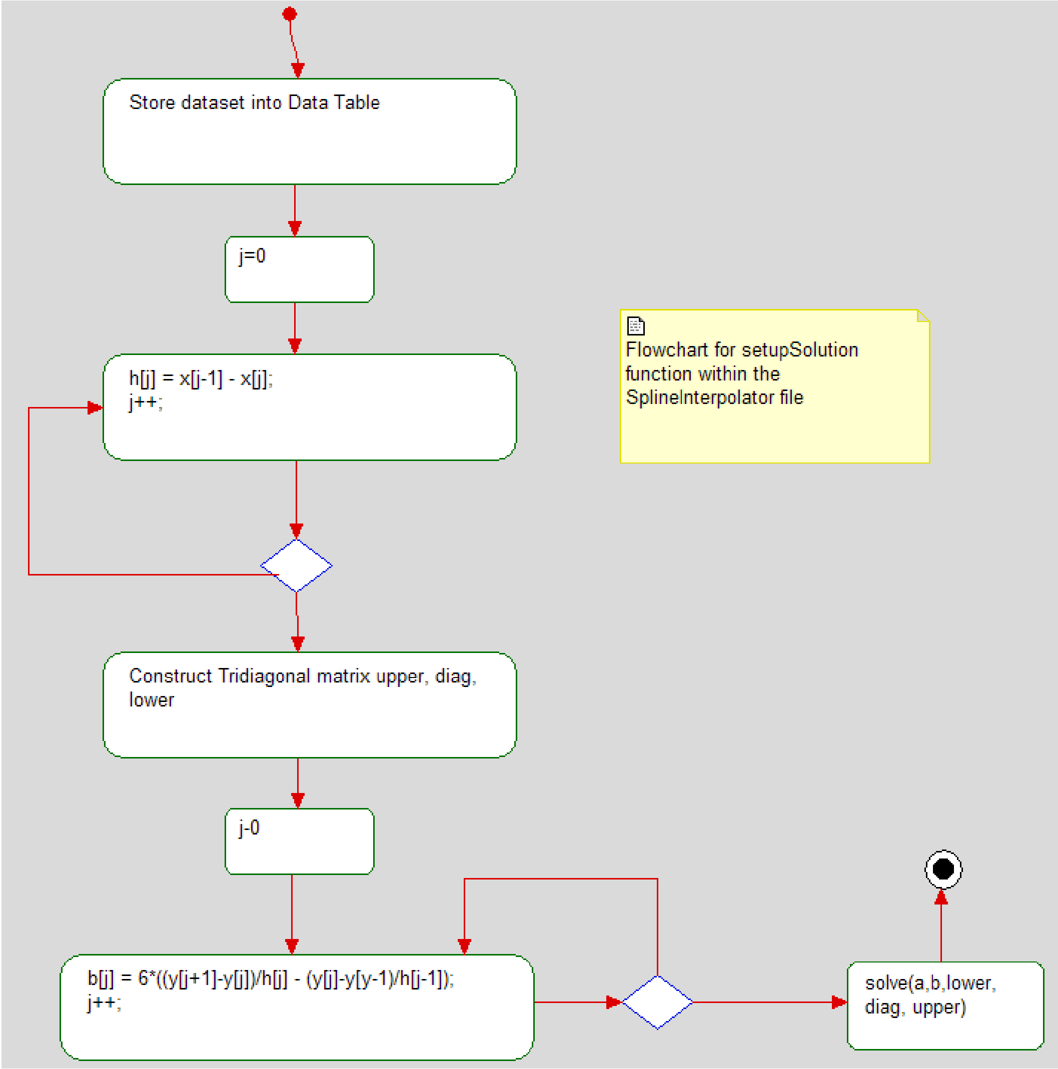


Execution and debugging w state machines & sequences



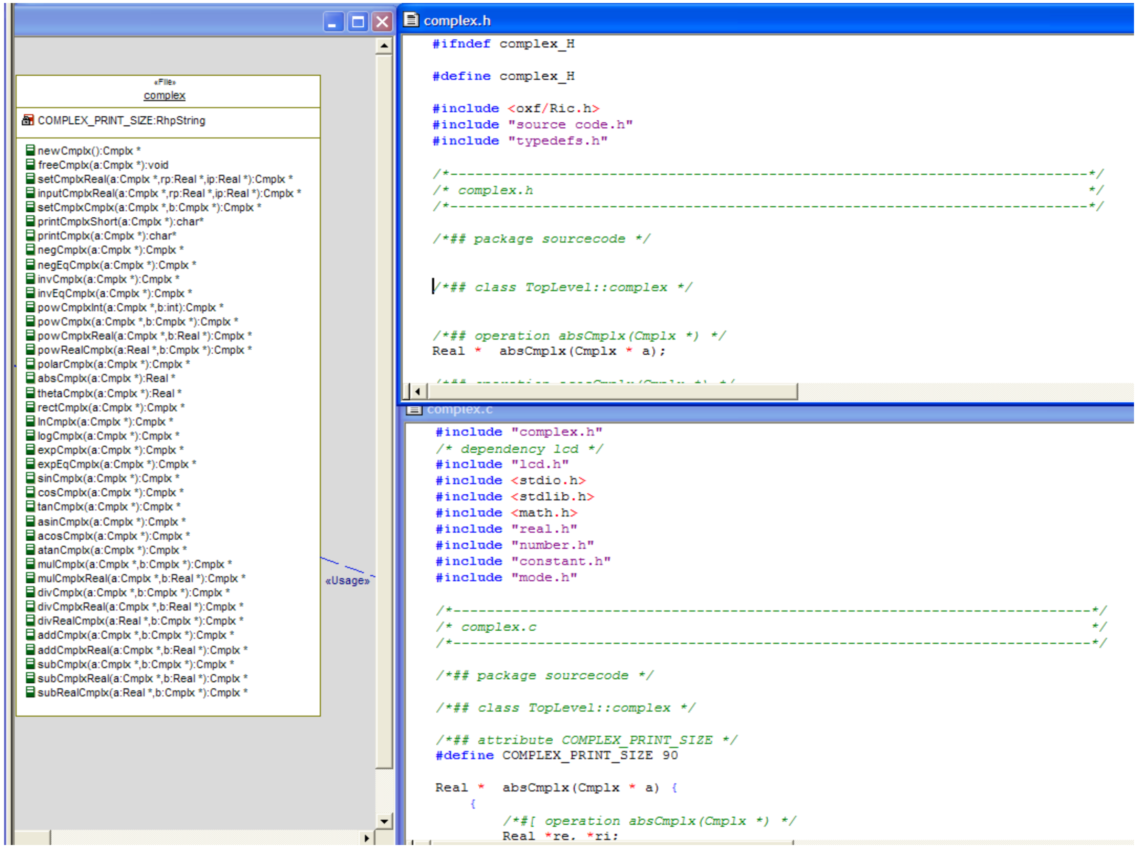
Flowchart

- Flowcharts (simplified activity diagrams) represent algorithms
  - Contain information contained within the Call Graph but adds
    - Sequence
    - Operators
      - Conditional
      - Fork
      - Join
- Most often, flow charts are assigned to functions
- With UML, you can
  - Execute and debug flow charts
  - Generate code from flow charts



Code View

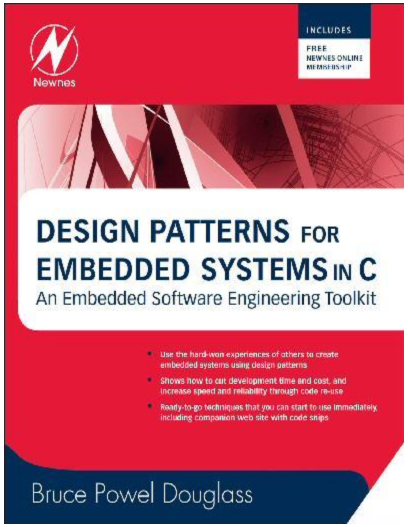
- Tools can maintain automatic synchronization between the code and the graphical views.
  - This is called dynamic model-code associativity or round-trip engineering
  - If you modify the code, the model changes
  - If you modify the model, the code changes
- You can also create a model from a source code base – this is known as Reverse Engineering

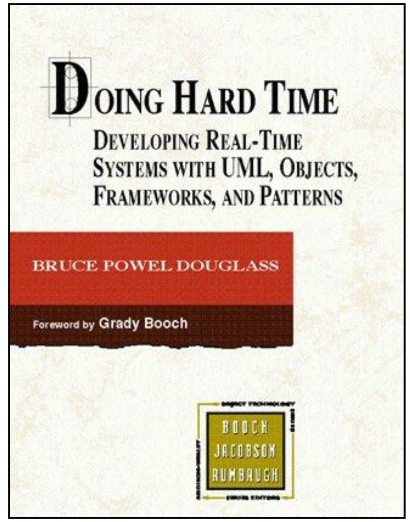


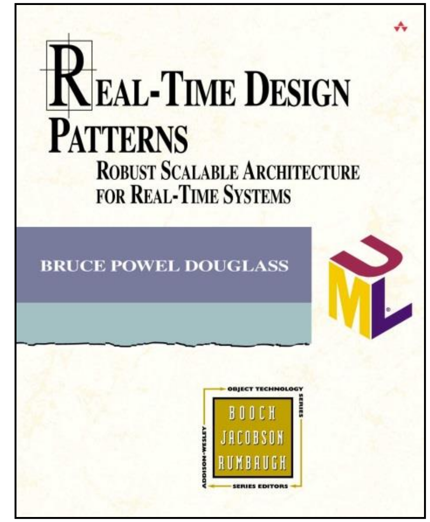
Summary

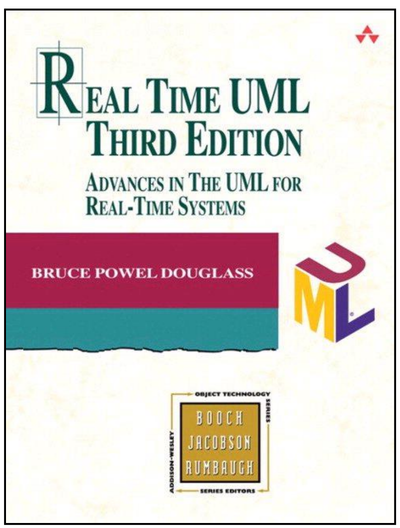
- Using a model-based approach provides real benefits for C developers
  - Improved understanding
  - Improved maintainability
  - Improved communication
  - Improved testability
  - Simplified compliance to safety standards
  - Automatic code generation
  - Ability to use existing legacy code
  - Ability to work within either the graphical or code views
- Using Graphical C allows the functional C developer to continue with a functional approach without requirement adoption of object oriented approaches
- Graphical C can be used to
  - document existing code, or
  - Develop new systems
- Graphical C can be extended as desired by adding UML and object oriented concepts downstream

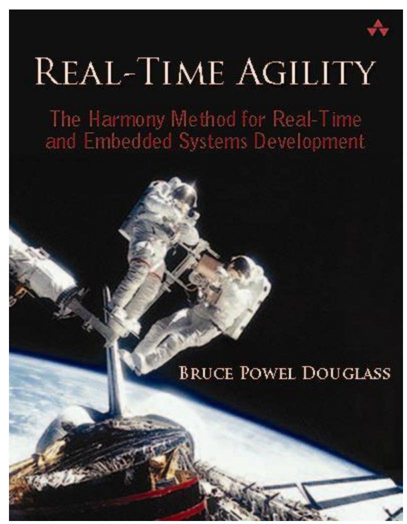
References

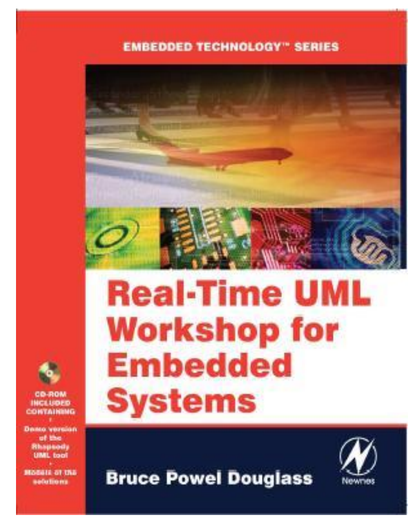

INCLUDES
FREE NEWNES ONLINE MEMBERSHIP
Newnes
DESIGN PATTERNS FOR EMBEDDED SYSTEMS IN C
An Embedded Software Engineering Toolkit
Bruce Powel Douglass


DOING HARD TIME
DEVELOPING REAL-TIME SYSTEMS WITH UML, OBJECTS, FRAMEWORKS, AND PATTERNS
BRUCE POWEL DOUGLASS
Foreword by Grady Booch
BOOCH
JACOBSON
RUMBAUGH


REAL-TIME DESIGN PATTERNS
ROBUST SCALABLE ARCHITECTURE FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS
UML
OBJECT TECHNOLOGY
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS


REAL TIME UML
THIRD EDITION
ADVANCES IN THE UML FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS
UML
OBJECT TECHNOLOGY
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS


REAL-TIME AGILITY
The Harmony Method for Real-Time and Embedded Systems Development
BRUCE POWEL DOUGLASS


EMBEDDED TECHNOLOGY™ SERIES
Real-Time UML Workshop for Embedded Systems
Bruce Powel Douglass
Newnes