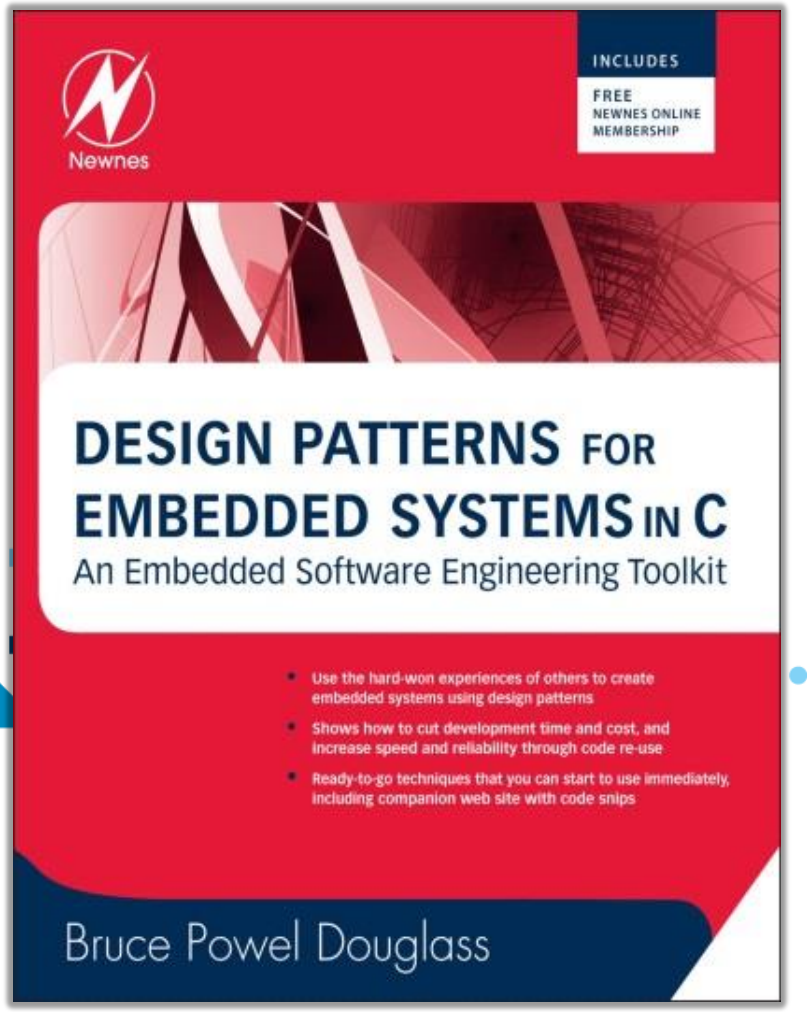
Design Patterns for Embedded Systems in C

Bruce Powel Douglass, Ph.D.
www.bruce-douglass.com
Bruce.Douglass@outlook.com
Twitter: @IronmanBruce

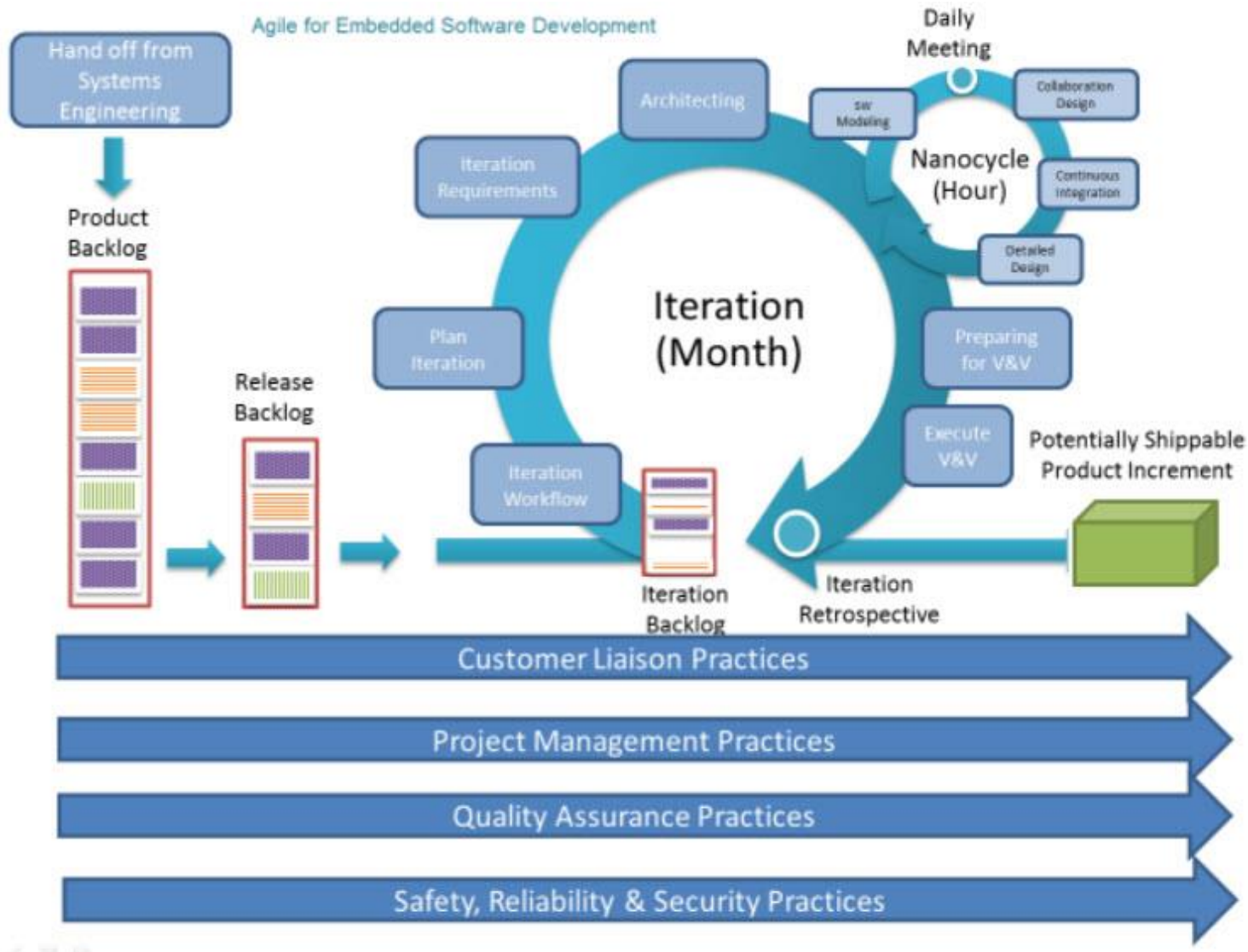


Analysis vs Design

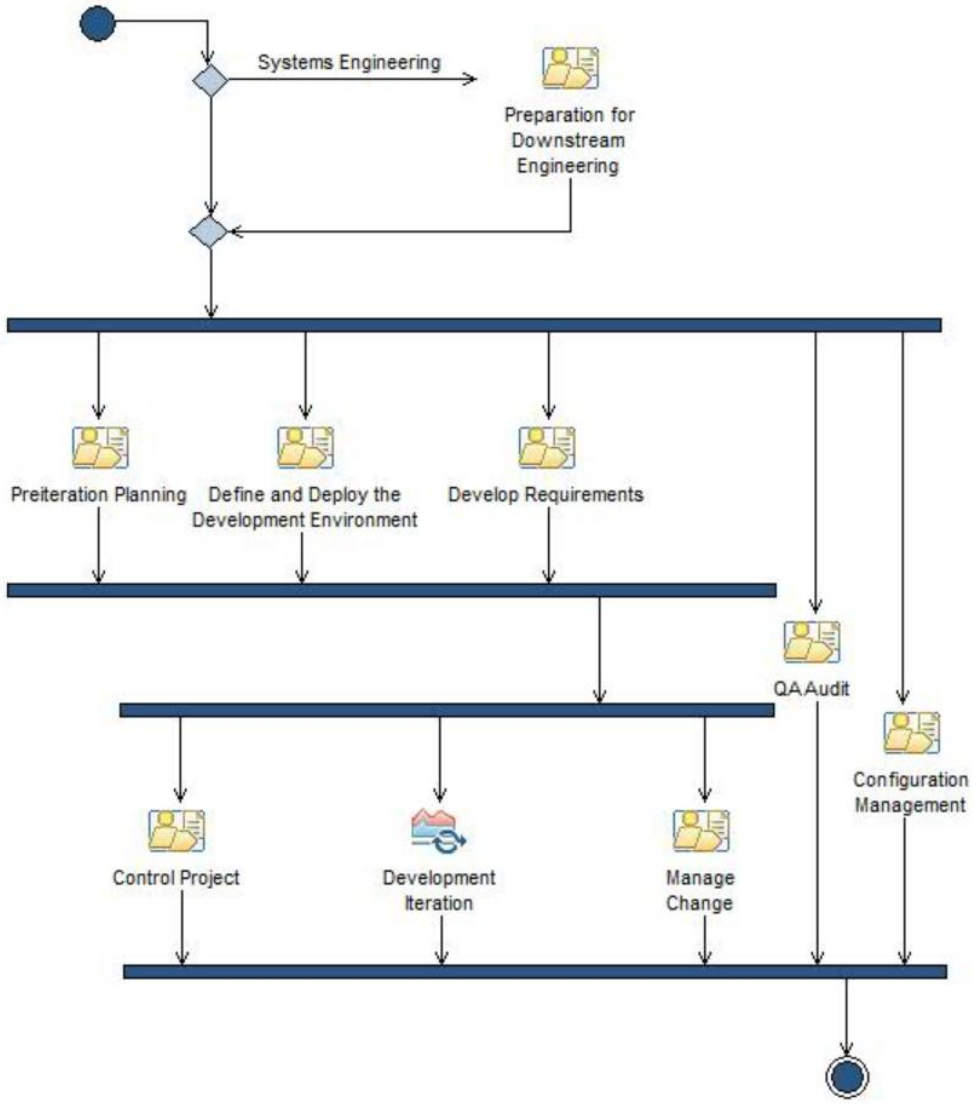
Analysis is the identification & specification of properties of a system that are *essential* for correctness

Design is the selection of one particular solution which optimizes the set of design criteria with respect to the relative importance of each

Harmony ESW Workflows

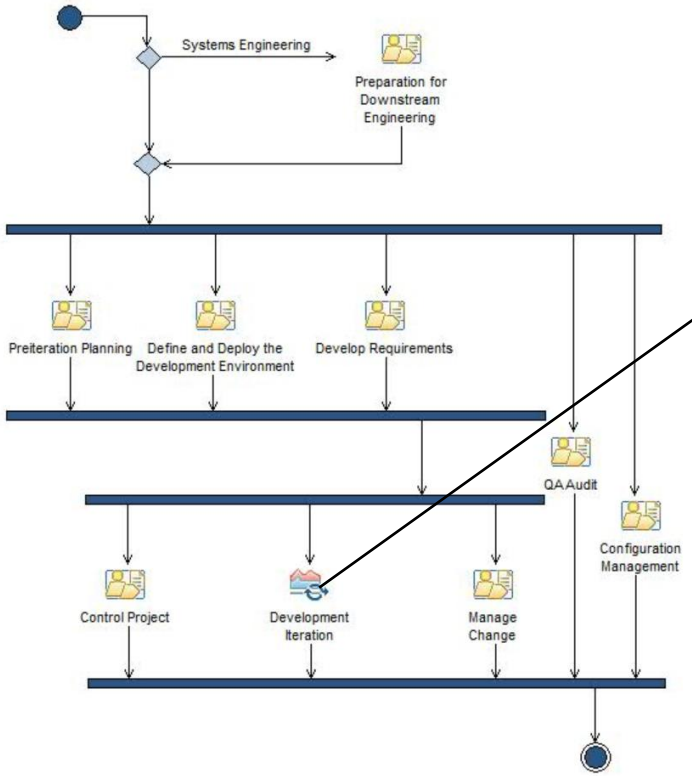


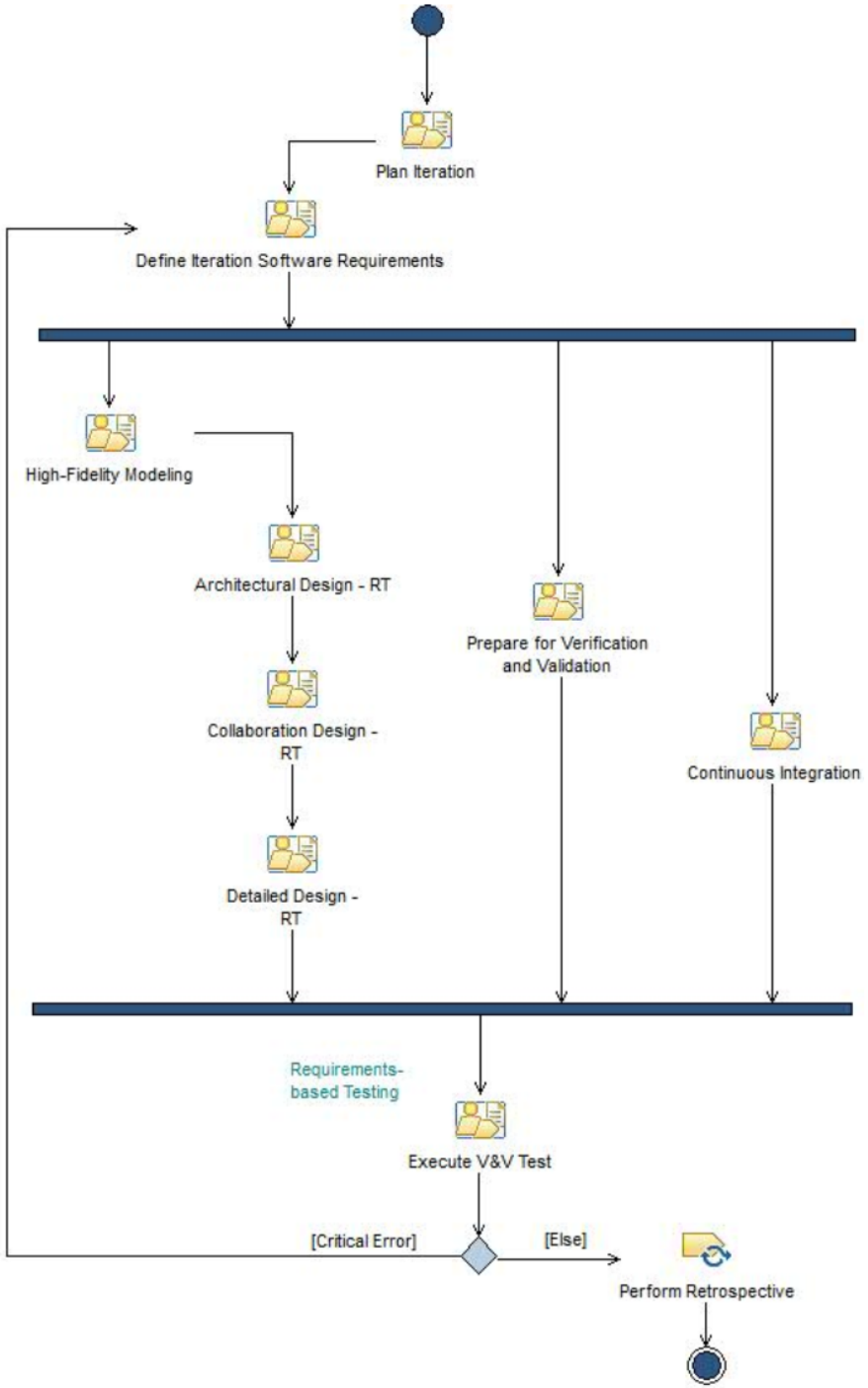
Harmony ESW Workflows



Development Iteration

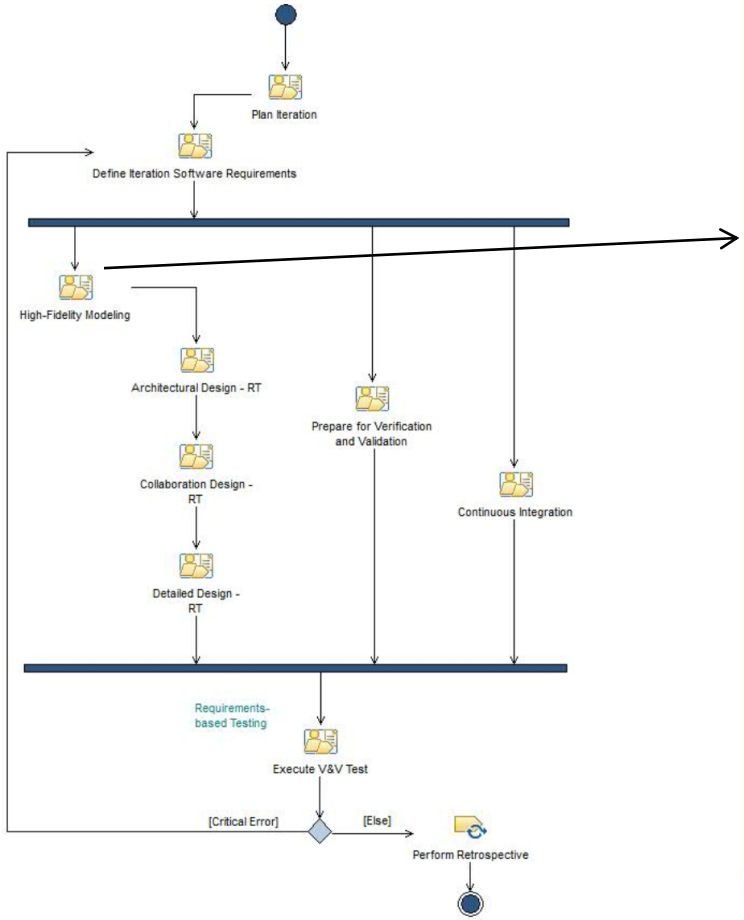
Produces models, source code, and resulting object software in an incremental fashion, typically every 1-4 weeks.

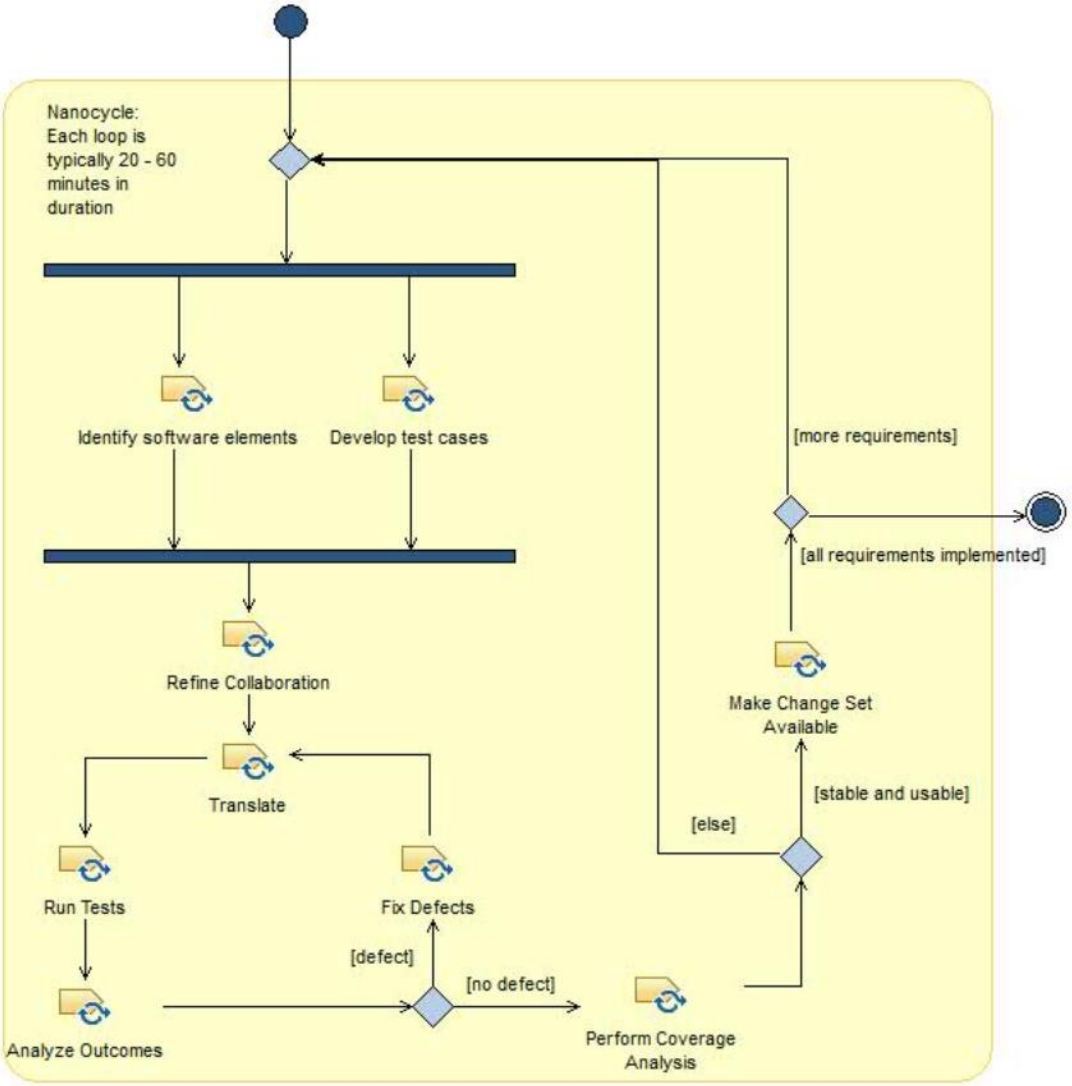




High-Fidelity Modeling

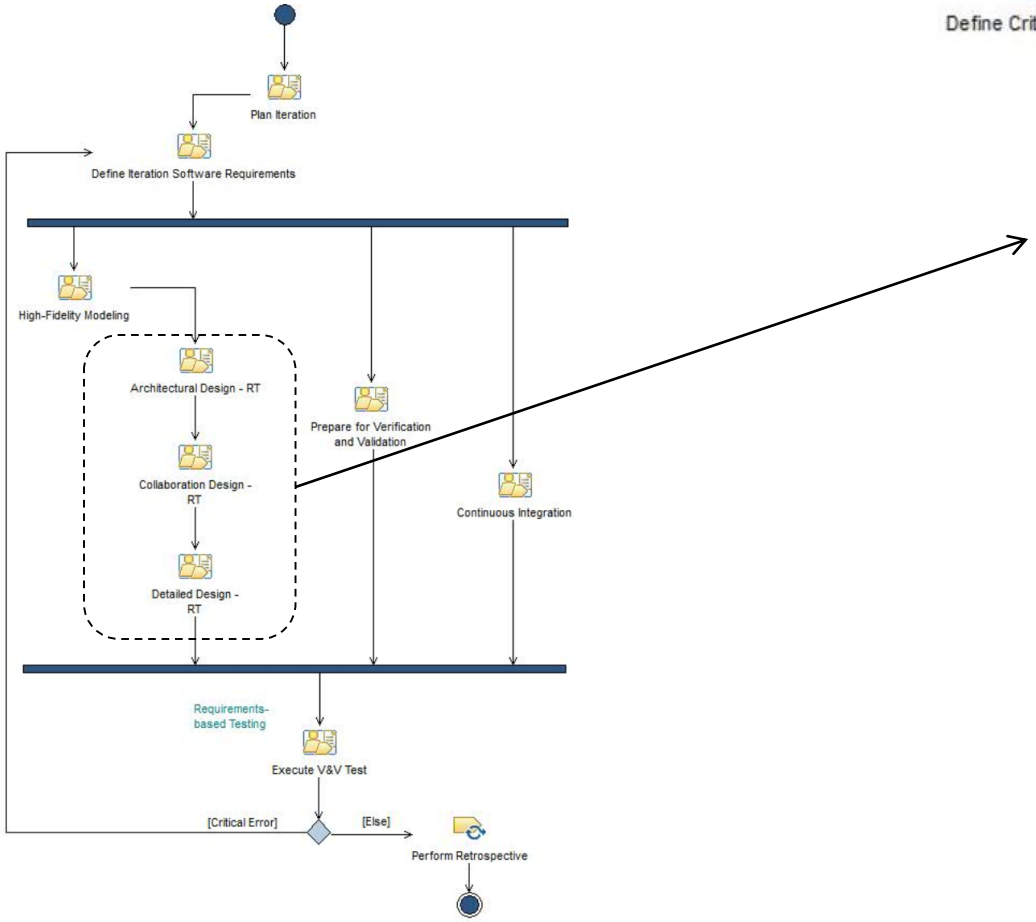
Produces an “analysis” model and code of the software: demonstrably functionally correct but not optimized

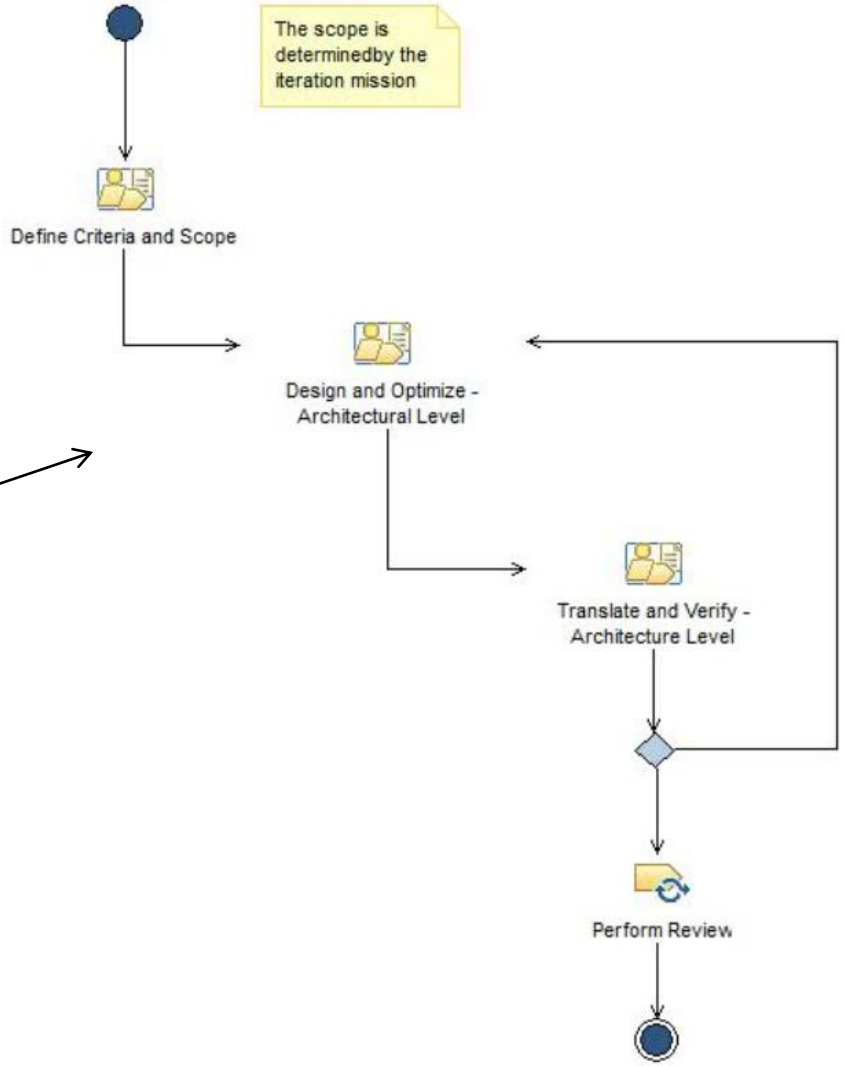




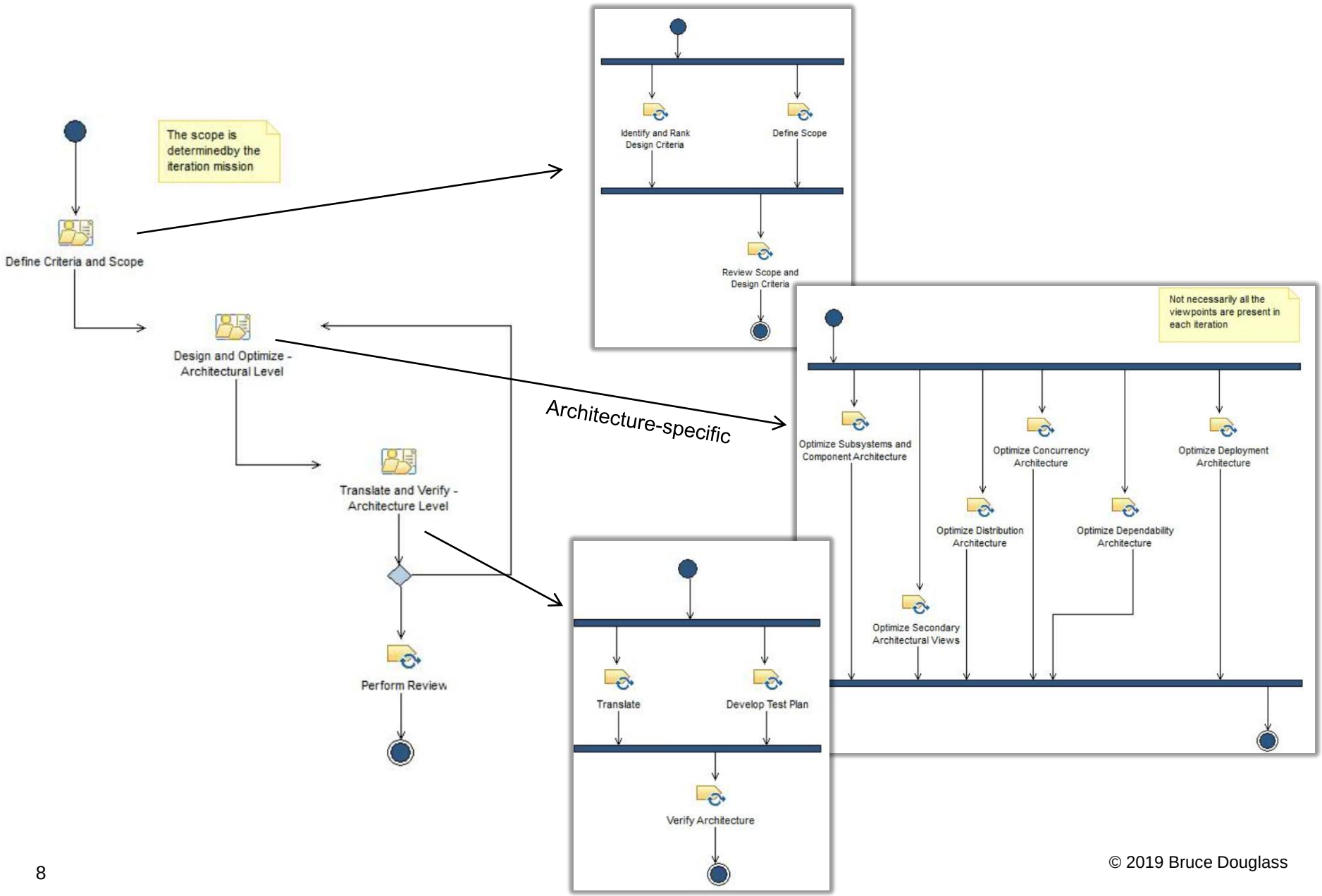
Design Workflows

Produces a “design” model and code of the software: demonstrably functionally correct and optimized against the selected criteria at the identified level of abstraction.





Design Workflows



Levels of Design

Optimization of individual functions, data structures, and classes

Detailed

Collaboration-wide optimization

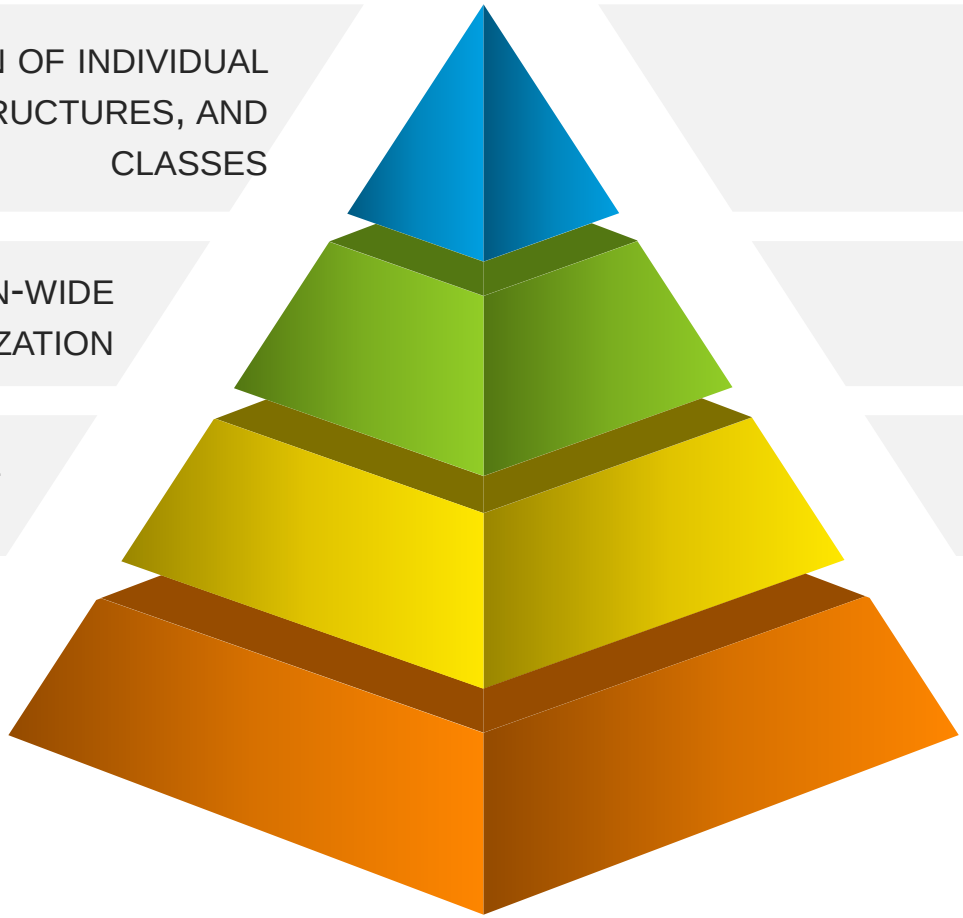
Collaboration

System-wide optimization

Architecture

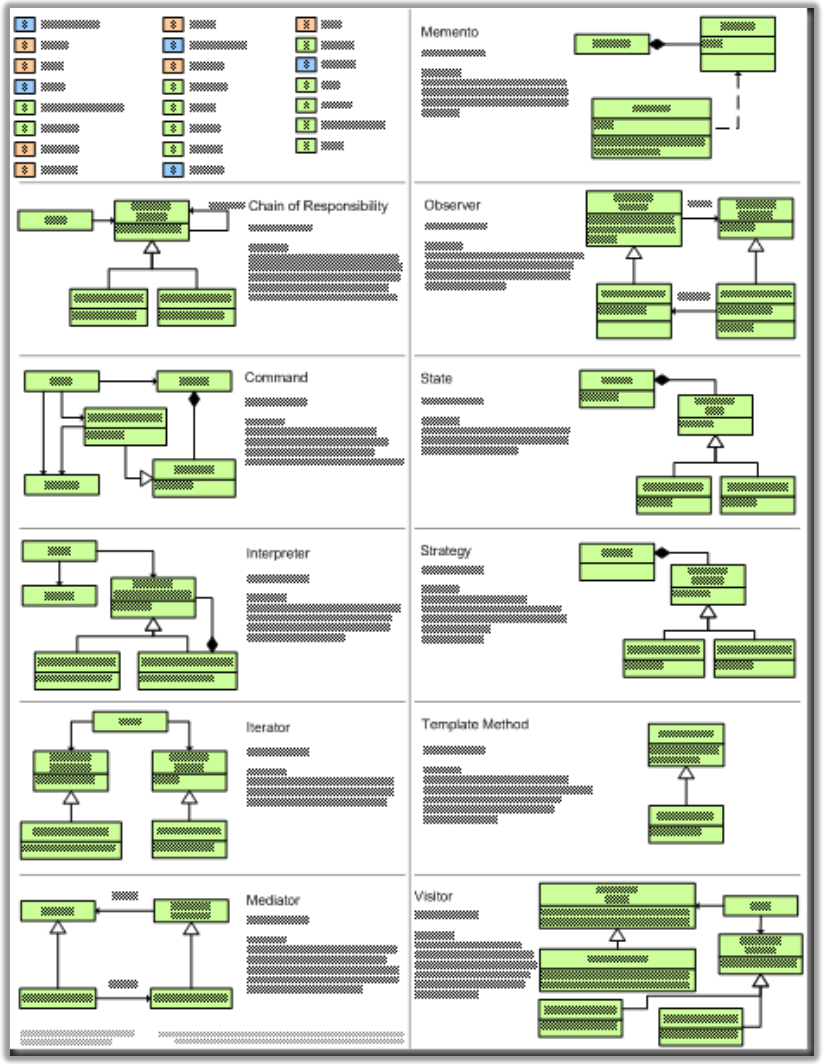
Functionally Correct

Analysis



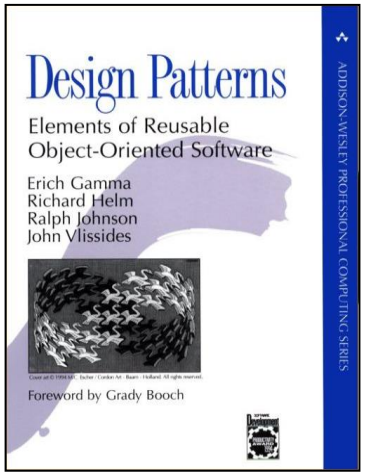
Common Design Optimization Criteria Drives Pattern Selection

- Simplicity
- Performance
  - Average
  - Worst-case
  - Predictability
  - Schedulability
- Resource usage
  - Robustness
  - Thread safety
  - Minimization of resources (space)
  - Minimization of resources (time)
  - Deadlock avoidance
- Safety
- Reliability
- Security
- Reusability
- Portability
- Extensibility & evolvability
- Maintainability
- Time-to-market
- Standard conformance



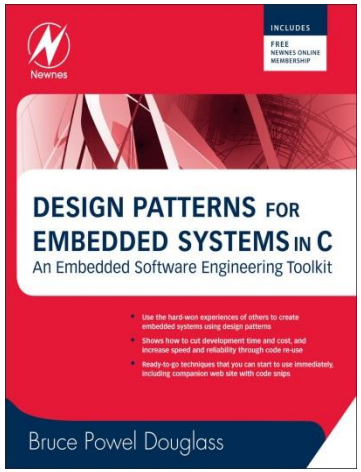
Design Patterns

- Design patterns are
  - generalized solutions to recurring optimization problems
  - Parameterized collaborations of objects, where the object roles are the *formal parameters* and the objects that play those roles are the *actual parameters* when the pattern is instantiated
- Patterns provide 2 kinds of elements
  - "Glue" classes that coordinate the elements work together
  - "Formal parameters" classes that are replaced by elements from the analysis model
- Important aspects
  - Applicability
  - What it optimizes
  - Solution
  - Consequences
    - Pros
    - Cons



"The classic text"

- GoF Categories:
  - Creational
  - Structural
  - Behavioral

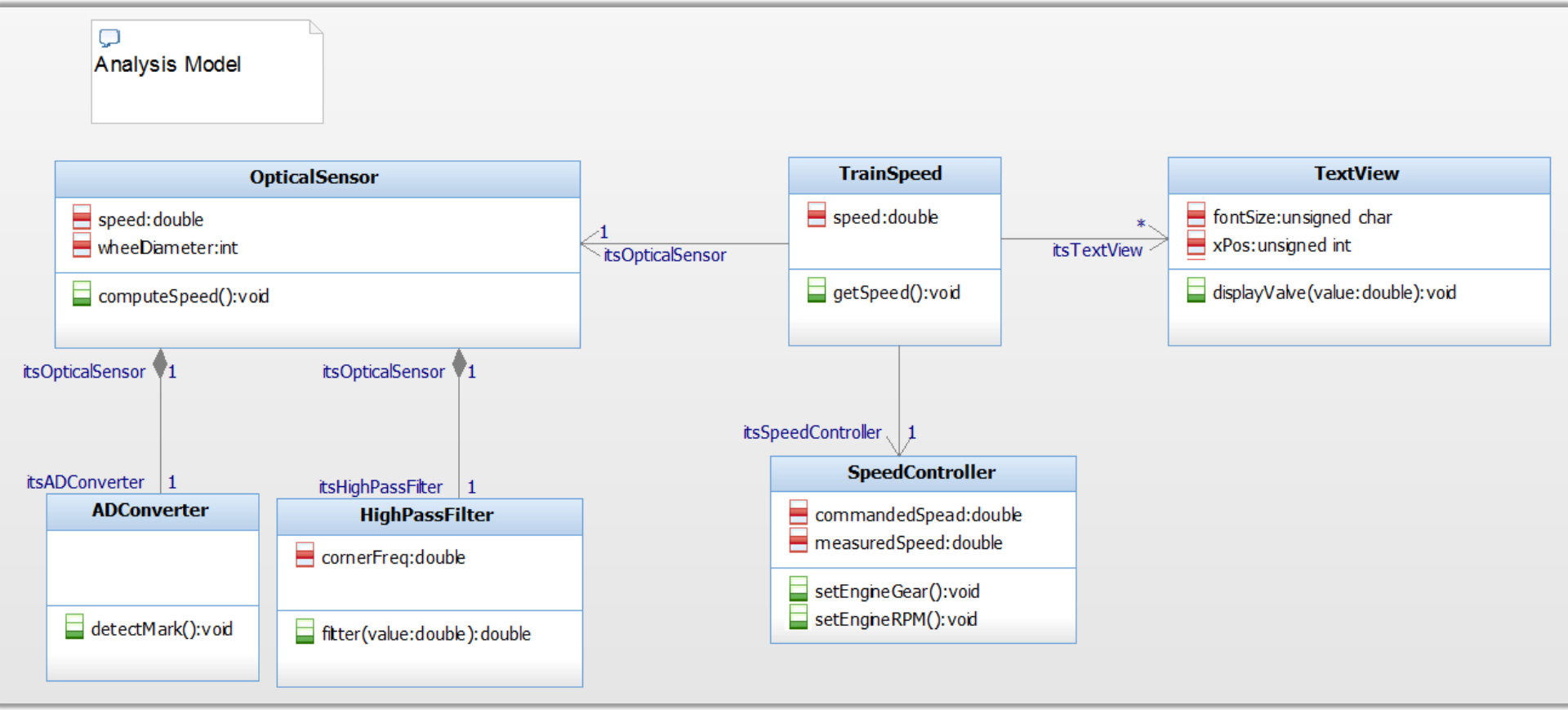


- Harmony Categories:
  - Creational
  - Structural
  - Behavioral
  - State Behavioral
  - Distribution
  - Dependability
  - Deployment
  - Concurrency & Resource

Example Analysis Model Collaboration

Optimization Criteria:

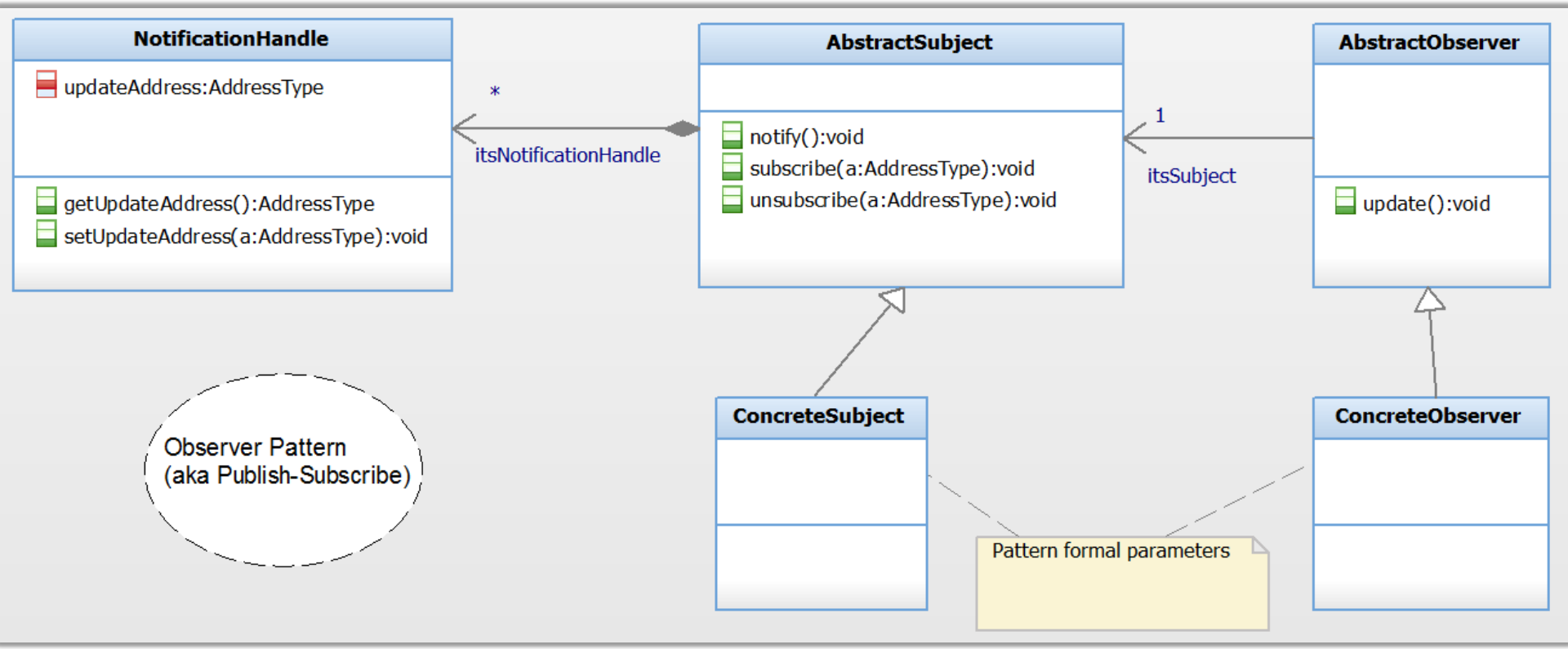
1. Ease of adding new **TextView** (and other) clients
2. Efficient use of bus bandwidth (only send data on bus when necessary)



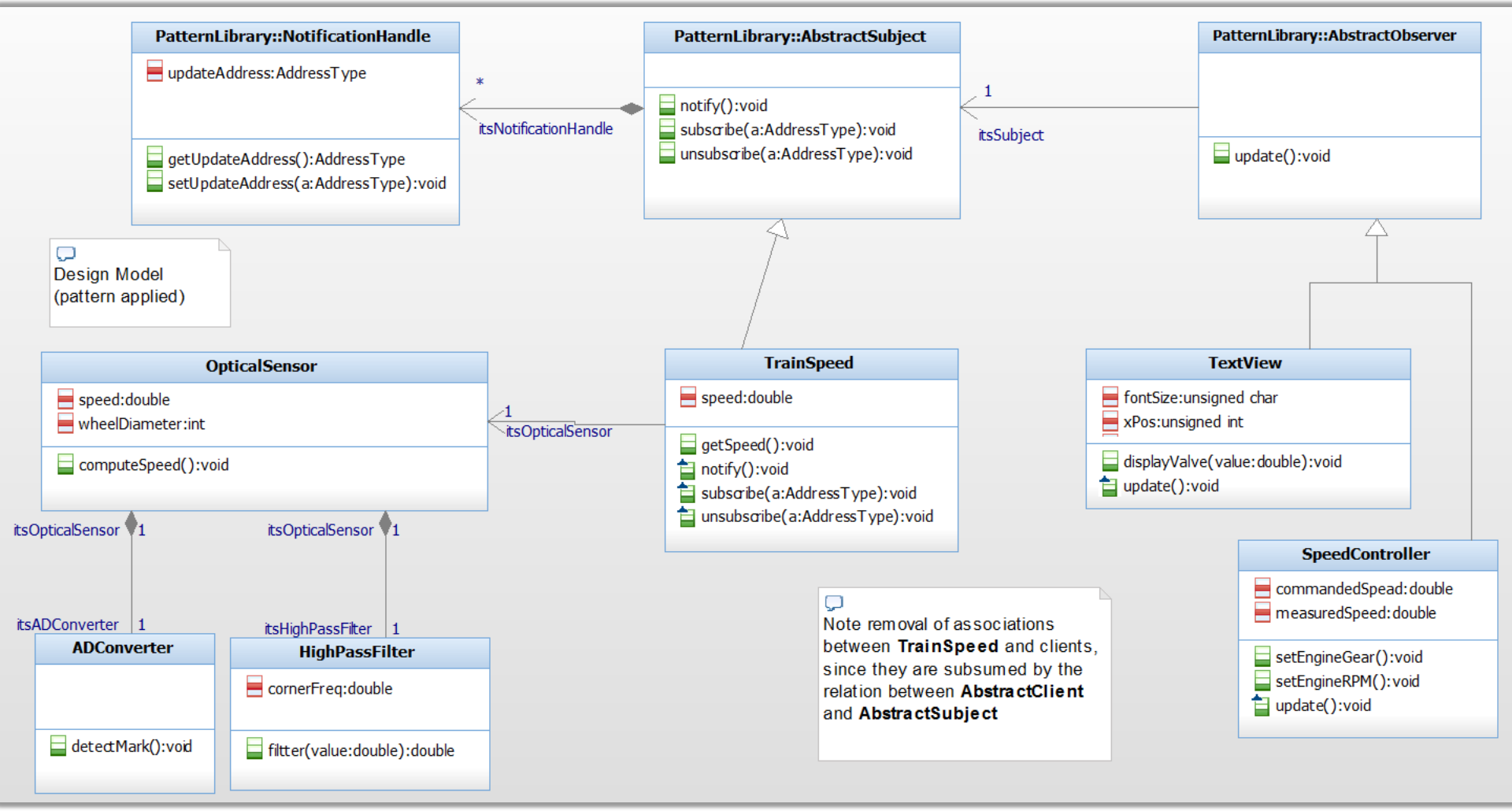
Selecting Patterns using Design Tradeoff Analysis

Design Solution	Design Criteria				Total Weighted Score
	Execution Efficiency	Maintainability	Run-time Flexibility	Memory Usage	
	Weight = 7	Weight = 5	Weight = 4	Weight = 7	
	Score	Score	Score	Score	
Client Server	3	7	8	5	123
Push Data	8	4	7	9	167
Observer Pattern	**8**	**7**	**9**	**9**	**190**

Design Pattern: Observer Pattern Specification



Design Pattern: Observer Pattern Instantiation

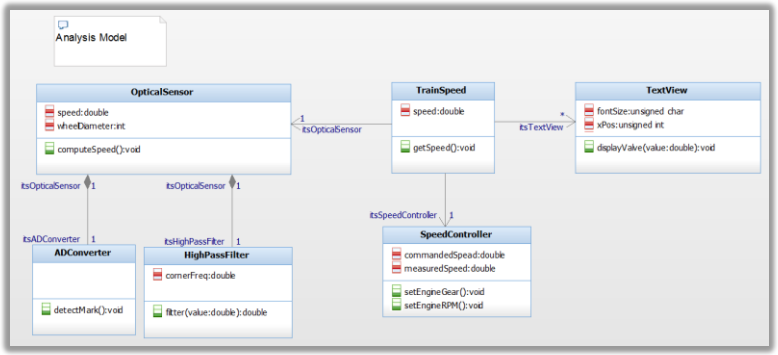


Why Use Design Patterns?

- Reuse effective design solutions
- Provide a more powerful vocabulary of design concepts to developers
- Develop “optimal” designs for specific design criteria
- Develop more understandable designs

How to Apply Design Patterns

- Construct a *analysis model* that *verifiably* meets the functional requirements
- This analysis model need not be optimized but it needs to be *verifiably correct*
  - This includes code generated from that design
- If any optimizations are needed, optimize for simplicity



How to Apply Design Patterns

- Identify the top 3-8 criteria (AKA Measures of Effectiveness (MOEs)) you want to optimize
- Rank them in order of importance from 1 (lowest) to 10 (highest)

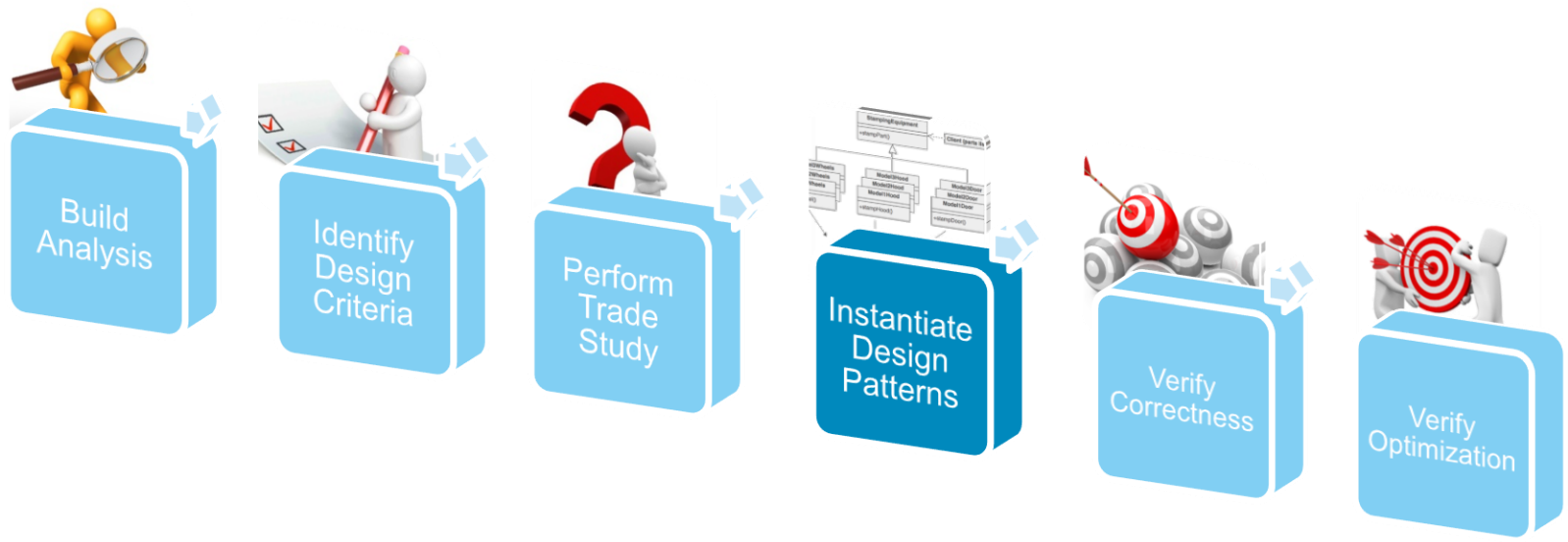
Criterion	Importance (1-10)
Execution efficiency	7
Maintainability	5
Run-time flexibility	4
Memory usage	7

How to Apply Design Patterns

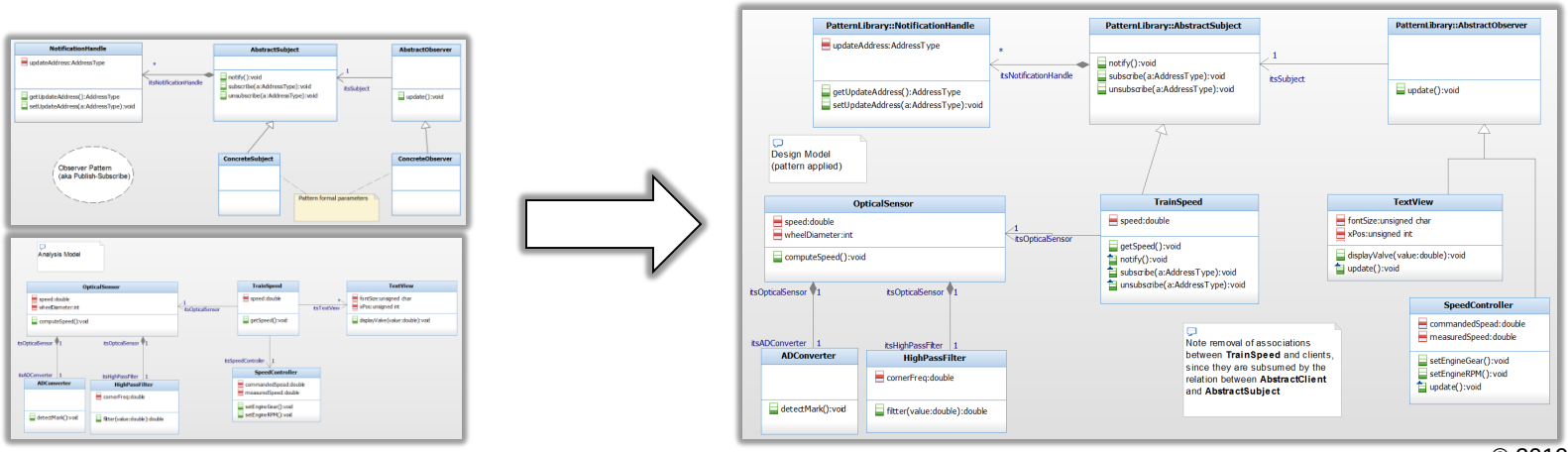
- Perform a weighted-sum analysis of the design alternatives
- Select the design patterns that best optimize your system overall

Design Solution	Design Criteria				Total Weighted Score
	Execution Efficiency	Maintainability	Run-time Flexibility	Memory Usage	
	Weight = 7	Weight = 5	Weight = 4	Weight = 7	
	Score	Score	Score	Score	
Client Server	3	7	8	5	123
Push Data	8	4	7	9	167
Observer Pattern	**8**	**7**	**9**	**9**	**190**

How to Apply Design Patterns

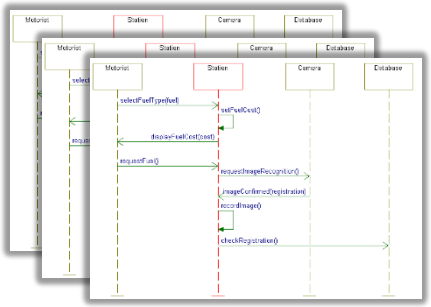


- Add in the design pattern, substituting your analysis elements for the parameters of the pattern and adding the glue elements to orchestrate their collaboration
- This is likely to require some refactoring

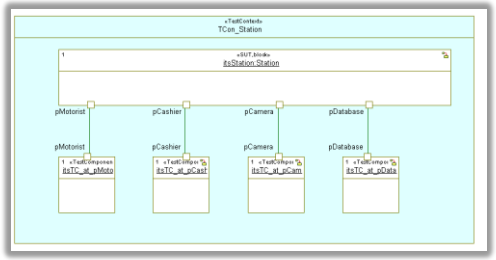


How to Apply Design Patterns

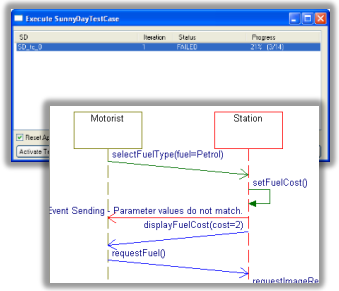
- *It used to work* (remember the "verifiably correct" aspect of the analysis model?)
- Verify that it still works



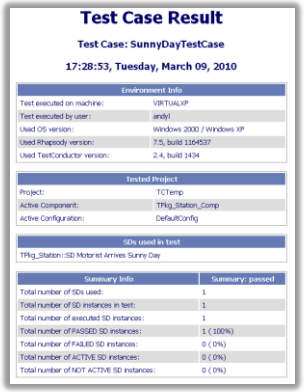
Test case



Test architecture



Test outcomes



Test verdicts

How to Apply Design Patterns

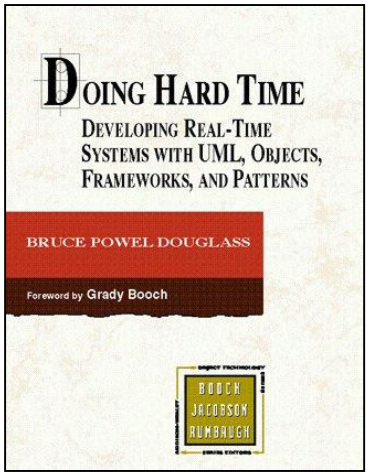
- You applied this design pattern for a reason. Did you achieve your goal?
  - Is the performance improvement adequate?
  - Is the memory usage small enough?
  - Is the system more maintainable?
  - Is the run-time flexibility adequate
  - Does it meet the safety / security / reliability goals?

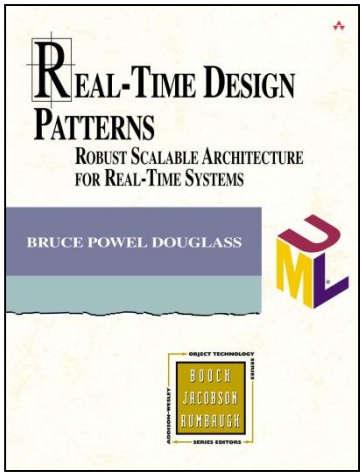
What's special about Embedded Systems?

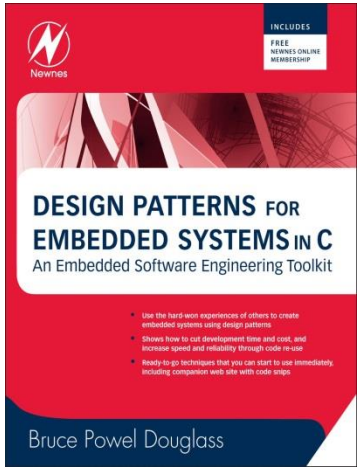
- Resource constraints
  - Most embedded systems are far more highly constrained in available memory, CPU cycles, and other resources
  - Embedded systems must often interface with custom hardware and create their own device drivers
  - Small embedded systems run with a bare bones RTOS or no OS at all
  - Most embedded systems are implemented in C
- Predictability and timeliness are often crucial to success
- Often require high dependability
  - Safety
  - Reliability
  - Security
- Design patterns for embedded applications provide reusable effective solutions to these concerns

Design Patterns for Embedded Systems

- Many categories are possible for design patterns for embedded systems. We'll use the following
  - Accessing hardware
  - Concurrency and resource management
  - State Machine implementation and usage
  - Safety, reliability, and security
  - Distribution and communications
  - Organization (e.g. layering) of subsystems
  - Reuse and product lines





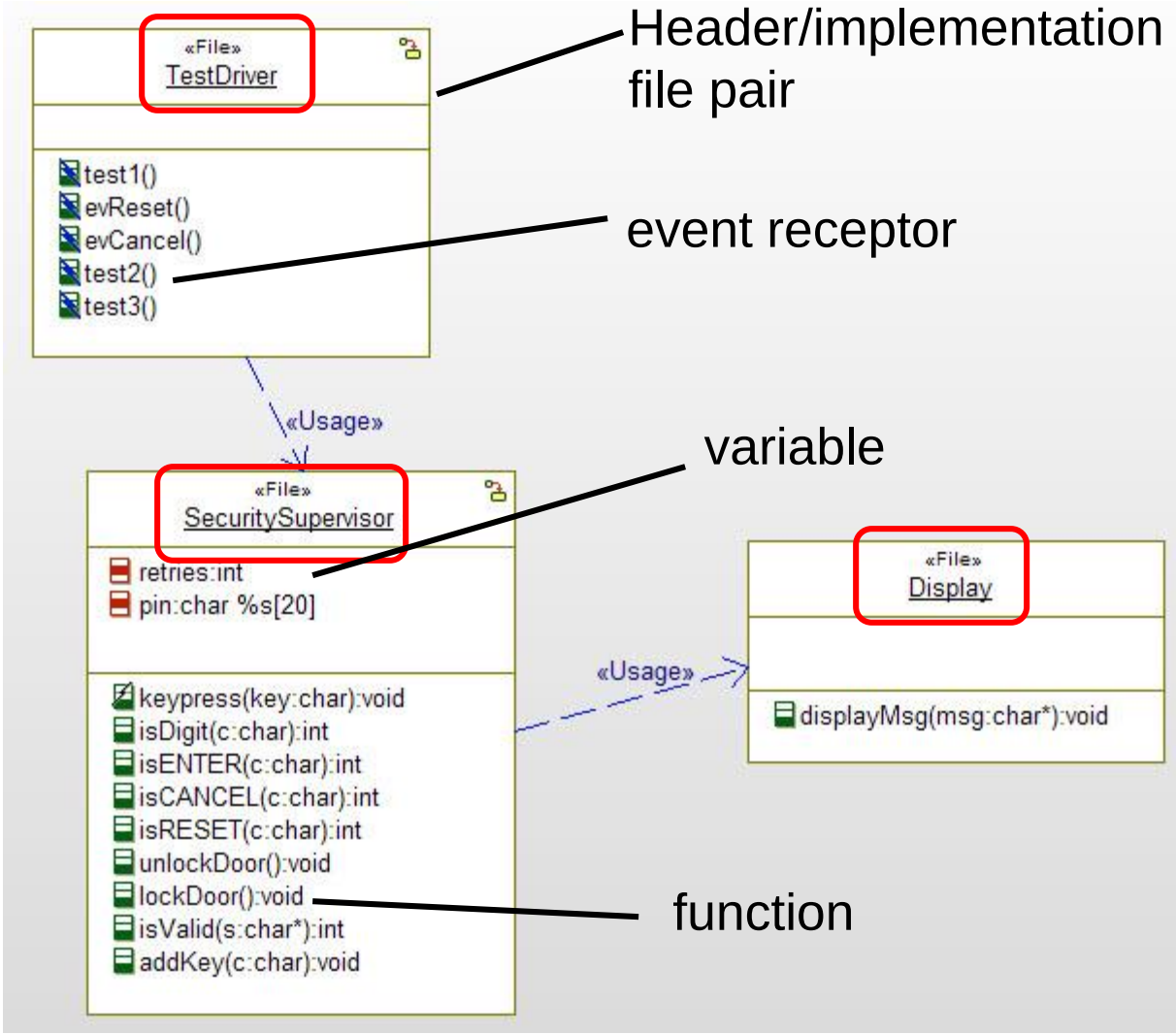


A note about design patterns in C

- Almost all design pattern books assure object oriented implementation in Java, C++, C# or similar languages
- Three styles for implementing patterns in C
  - File-based
    - This is “standard” C in which the application source code is organized into pairs of files (header and implementation). Not all common patterns are easy to implement in this style. This style organizes files around “class” concept but it’s all vanilla C.
  - Object-based
    - This approach uses **structs** to represent the classes (instances of which comprise the objects) and manually name mangled functions manipulate the data stored in the struct. Especially useful when there will multiple instances (variables) of a class or type.
  - Object-oriented
    - This style is similar to object-based except that the struct itself contains **function pointers** as a means to implement polymorphism and virtual functions, something required for implementing some patterns (those that require inheritance or polymorphism)

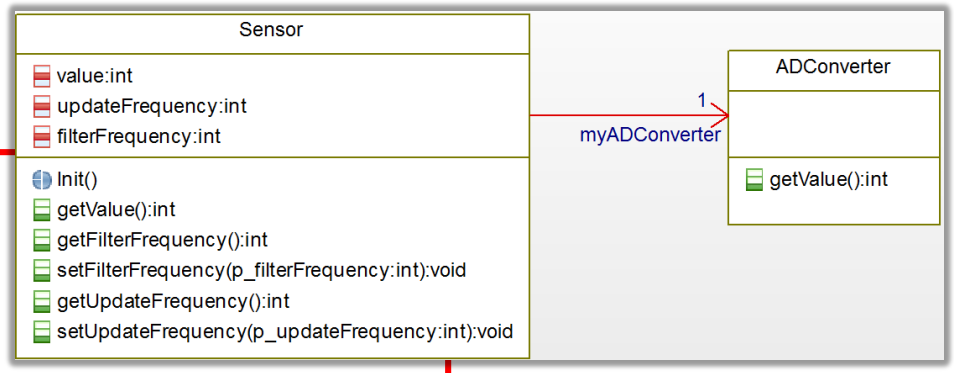
Classes represented as Files in C

- This is just a way to graphically visualize “standard C”
- A file pair (*.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”



C Object Based Design (header file)

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




```
#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)

```
#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* const me, int
p_updateFrequency) {
    me->updateFrequency = p_updateFrequency;
}

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

```
        /* 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 Oriented Design (header file)

- The function pointers support polymorphism and virtual functions

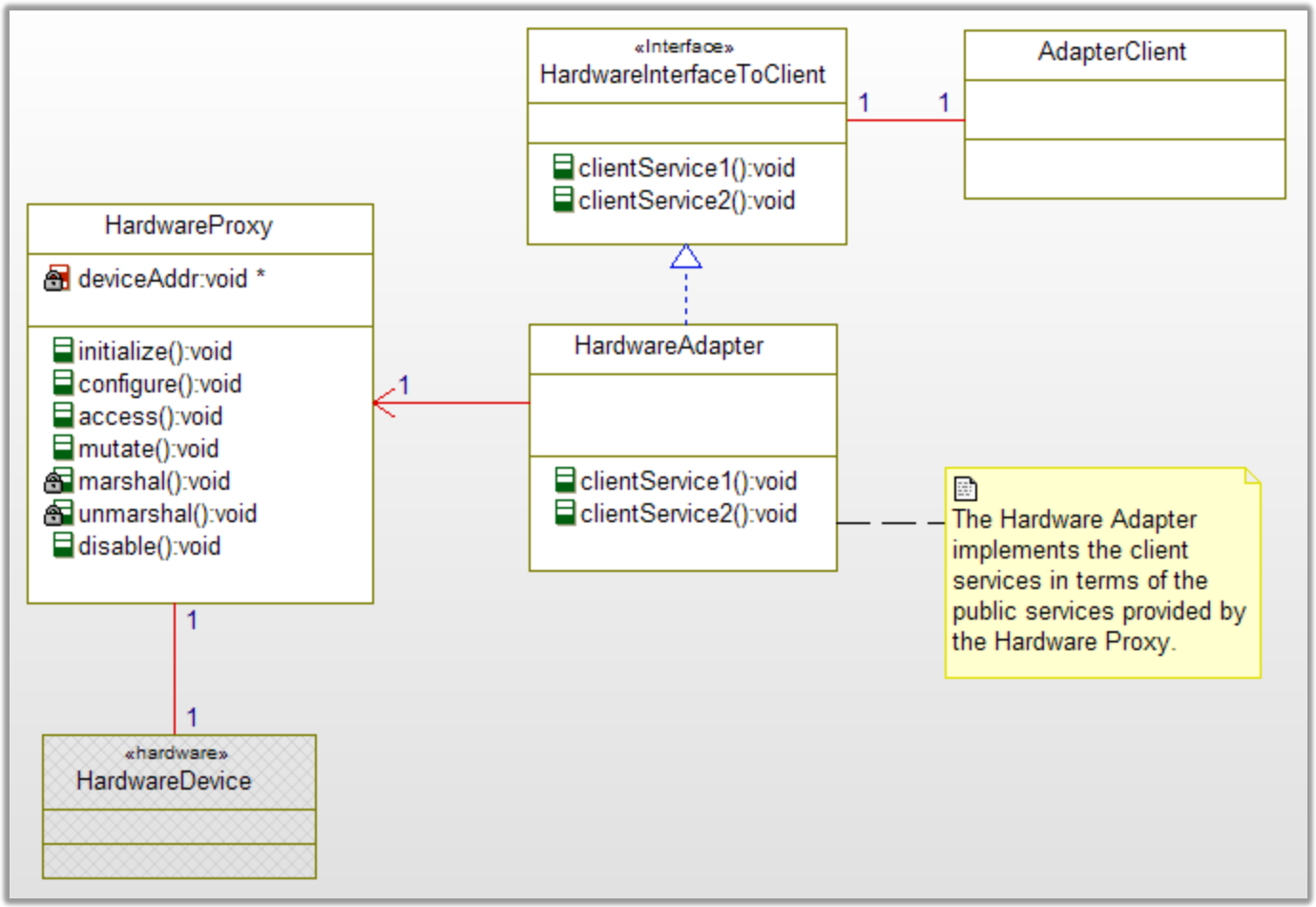
```
#ifndef Sensor_H
#define Sensor_H
#include "ADConverter.h"

    /* 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 = getFilterFrequency;
    me->setFilterFreq = setFilterFrequency;
}
```

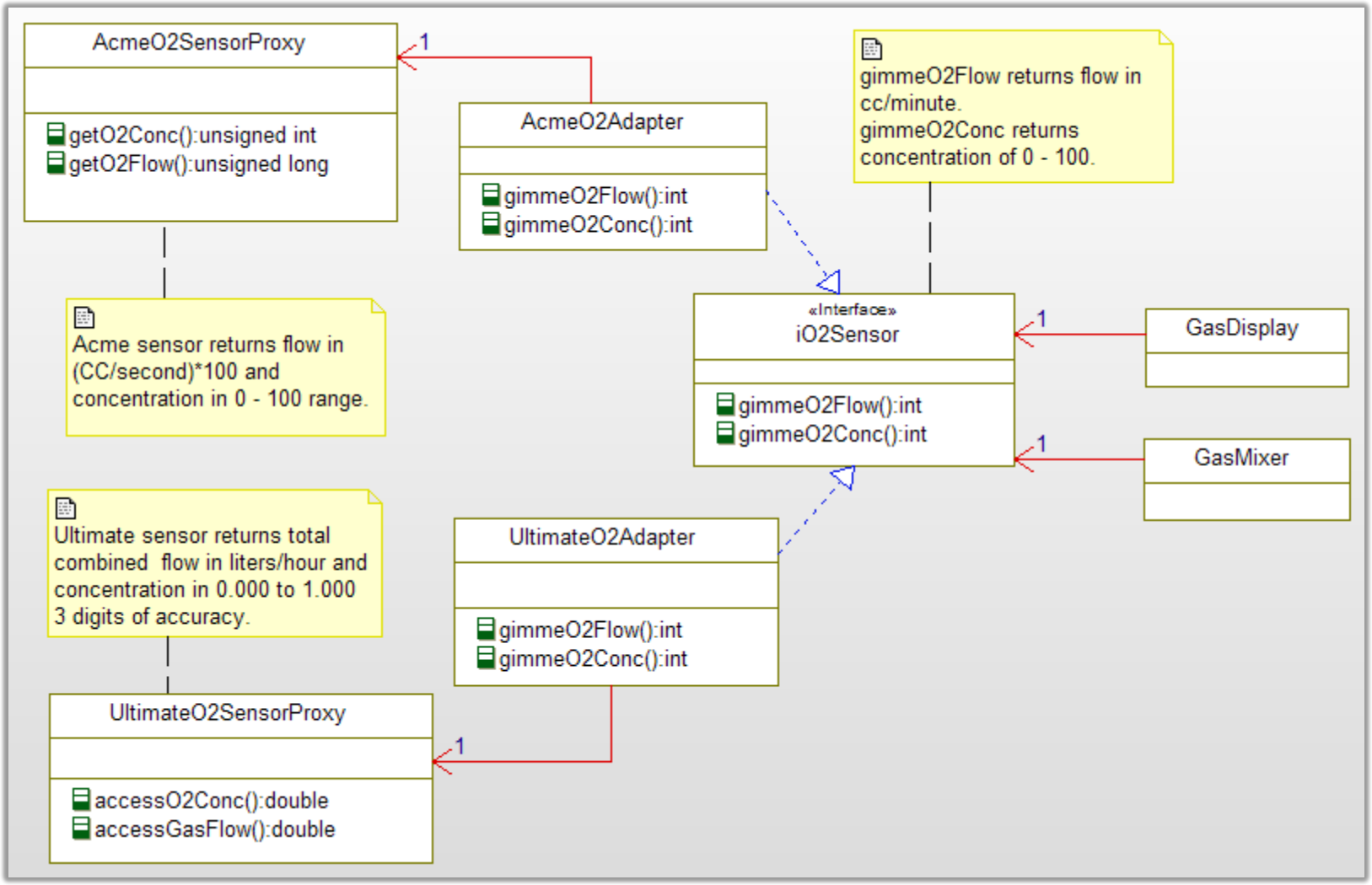
Pattern: Hardware Adapter

- Abstract
  - The Hardware Adapter Pattern is useful when the application requires or uses one interface, but the actual hardware provides another. The pattern creates an element that converts between the two interfaces.
- Problem
  - While hardware that performs similar functions tend to have similar interfaces, often the information they need and the set of services differ. Rather than rewrite the clients of the hardware device to use the provided interface, an adapter is created that provides the expected interface to the clients while converting the requests to and from the actual hardware interface.
- Solution
  - Create a class that performs that mapping of the actual interface (provided by the Hardware Proxy) and the interface required by the client
- Consequences
  - Allows reuse of existing hardware proxies in different applications and for different clients without rewriting that software.
  - This pattern adds a level of indirection and therefore some run-time performance overhead

Pattern: Hardware Adapter



Example: Hardware Adapter



Sample Code: Hardware Adapter

```
int AcmeO2Adapter_gimmeO2Conc(AcmeO2Adapter* const me) {
return me->itsAcmeO2SensorProxy->getO2Conc();
}
int AcmeO2Adapter_gimmeO2Flow(AcmeO2Adapter* const me) {
return (me->itsAcmeO2SensorProxy->getO2Flow()*60)/100;
}
```

AcmeO2Adapter.c

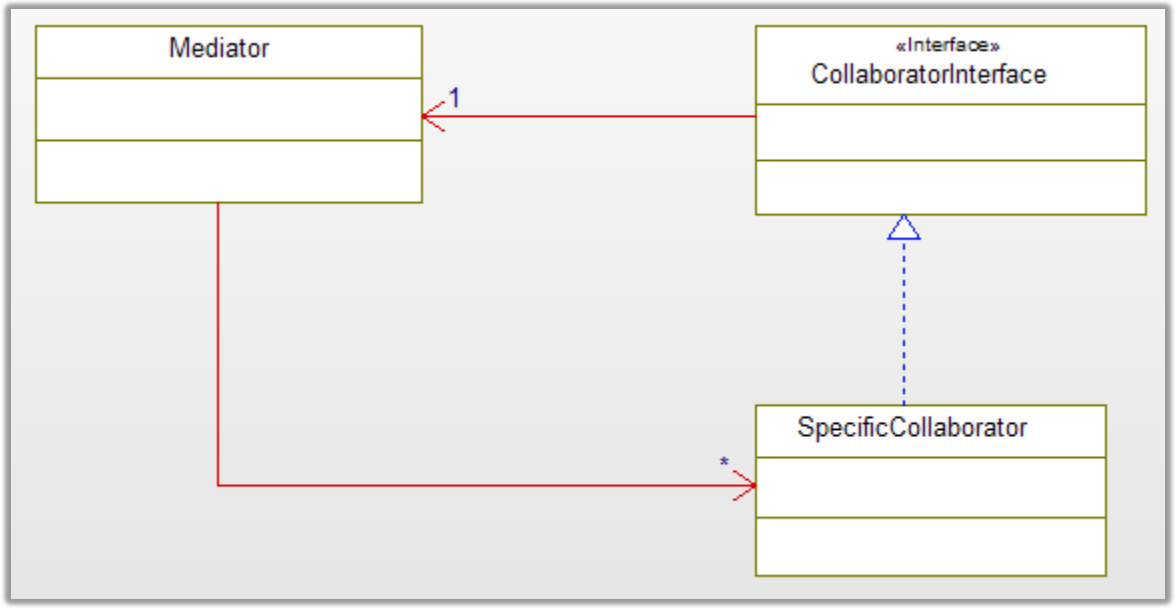
```
int UltimateO2Adapter_gimmeO2Conc(UltimateO2Adapter* const me) {
return int(me->getItsUltimateO2SensorProxy->accessO2Conc()*100);
}
int UltimateO2Adapter_gimmeO2Flow(UltimateO2Adapter* const me) {
double totalFlow;
// convert from liters/hr to cc/min
totalFlow = me->itsUltimateO2SensorProxy->accessGasFlow() * 1000.0/60.0;
// now return the portion of the flow due to oxygen and return it as an integer
return (int)(totalFlow * me->itsUltimateO2SensorProxy->accessO2Conc());
}
```

UltimateO2Adapter.c

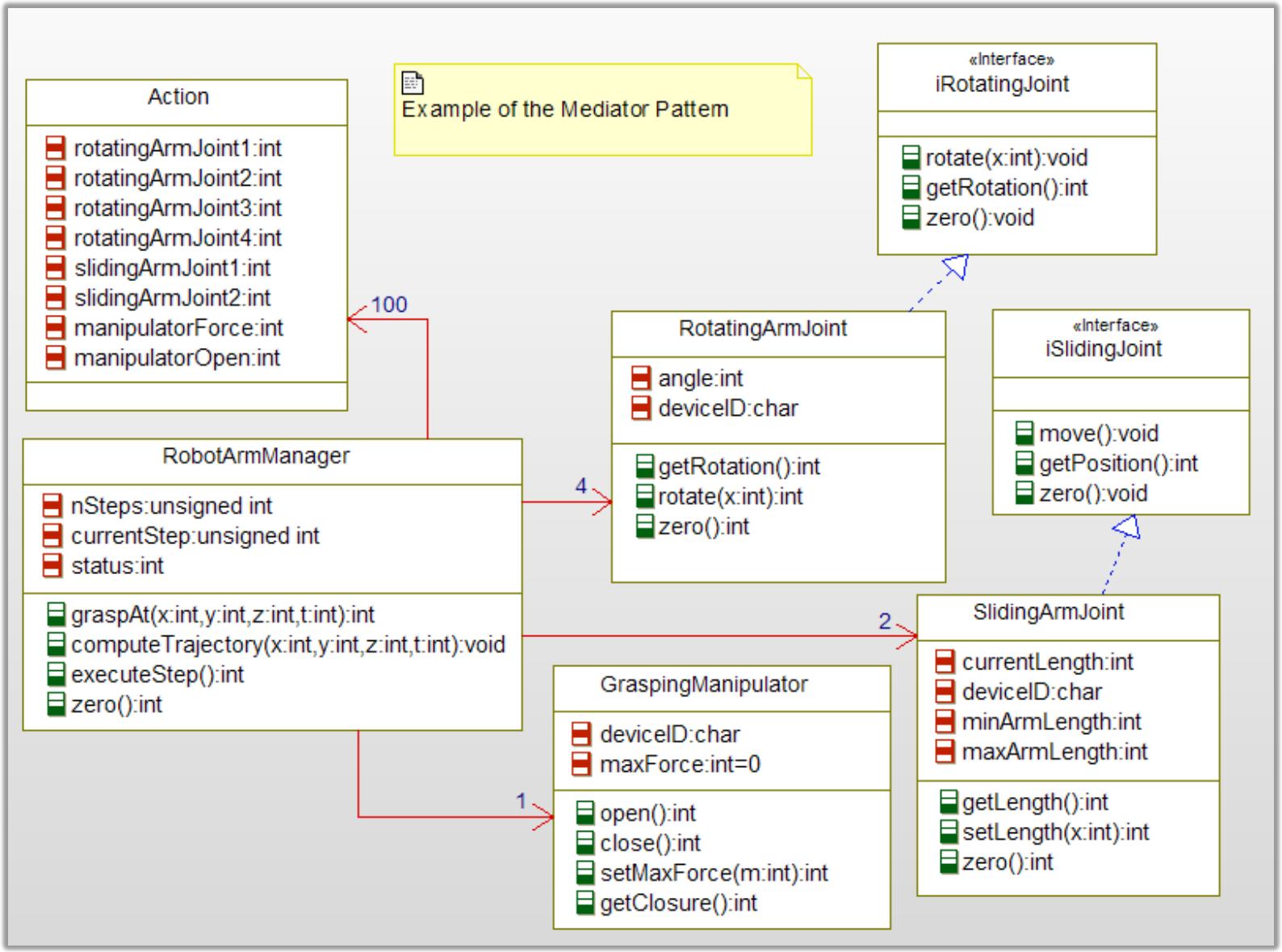
Pattern: Mediator

- Abstract
  - The Mediator Pattern is particularly useful for managing different hardware elements when their behavior must be coordinated in well-defined but complex ways. It is particularly useful for C applications because it doesn't require a lot of specialization (subclassing), which can introduce its own complexities into the implementation.
- Problem
  - Many embedded applications control sets of actuators that must work in concert to achieve the desired effect. For example, to achieve a coordinated movement of a multi-joint robot arm, all the motors must work together to provide the desired arm movement. Similarly, using reaction wheels or thrusters in a spacecraft in three dimensions requires many different such devices acting at precisely the right time and with the right amount of force to achieve attitude stabilization.
- Solution
  - The Mediator Pattern uses a mediator class to coordinate the actions of a set of collaborating devices to achieve the desired overall effect. The Mediator class coordinates the control of the set of multiple Specific Collaborators. Each Specific Collaborator must be able to contact the Mediator when an event of interest occurs.
- Consequences
  - This pattern creates a mediator that coordinates the set of collaborating actuators but without requiring direct coupling of those devices. This greatly simplifies the overall design by minimizing the points of coupling and encapsulating the coordination within a single element.
  - Since many embedded systems must react with high time fidelity, delays between the actions may result in unstable or undesirable effects. It is important that the mediator class can react within those time constraints.

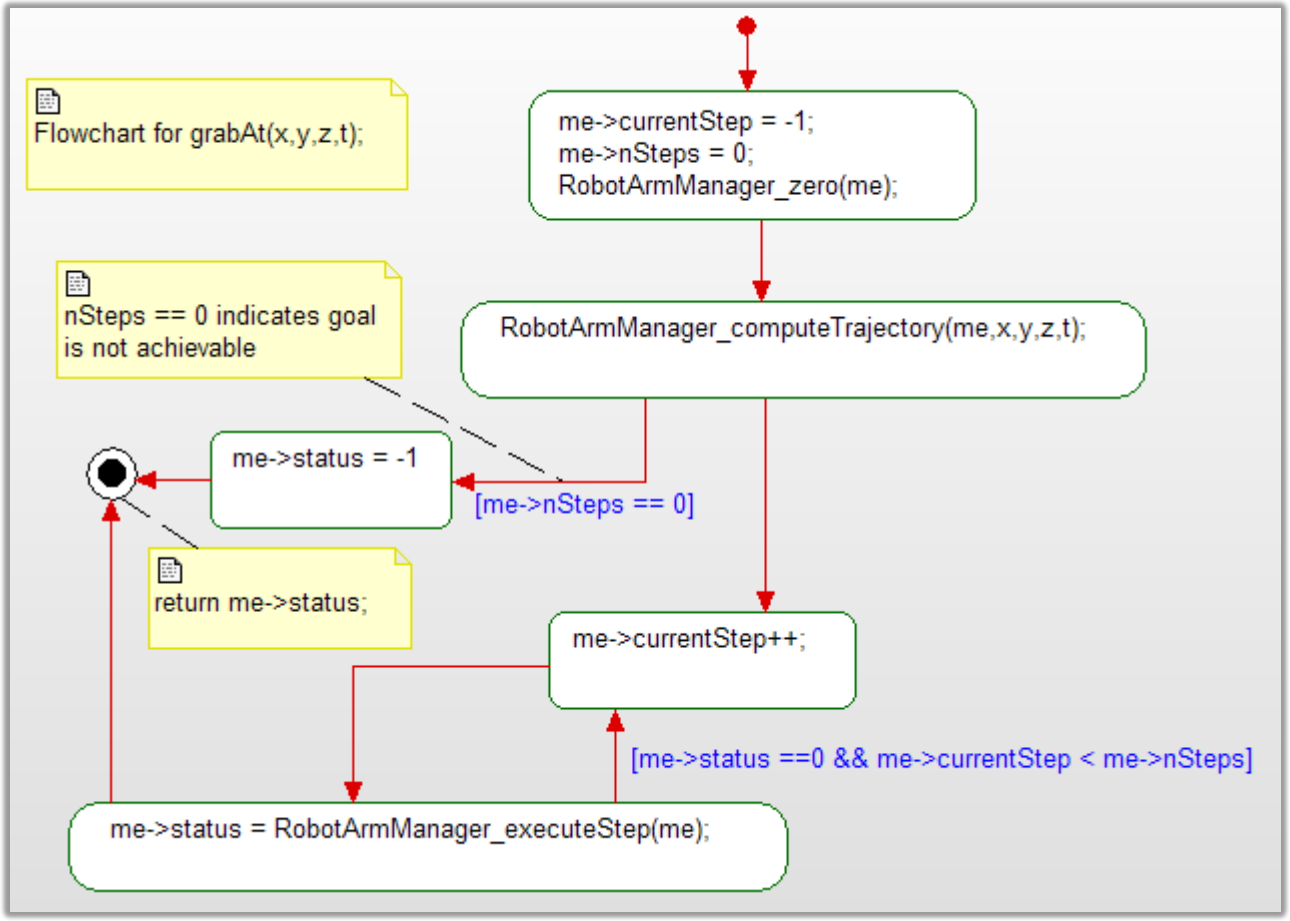
Pattern: Mediator



Example: Mediator



Example: Mediator



Source Code: Mediator (RobotArmManager.h)

```
#ifdef ROBOTARMMANAGER_H
#define ROBOTARMMANAHER_H
#include "GraspingManipulator.h"
#include "RotatingArmJoint.h"
#include "SlidingArmJoint.h"
#include "Action.h"
/* class RobotArmManager */
typedef struct RobotArmManager RobotArmManager;
struct RobotArmManager {
    unsigned int currentStep;
    unsigned int nSteps;
    struct GraspingManipulator* itsGraspingManipulator;
    struct RotatingArmJoint *itsRotatingArmJoint[4];
    struct SlidingArmJoint *itsSlidingArmJoint[2];
    struct Action *itsAction[100]; /* set of actions to perform to execute the planned trajectory */
    int status;
};
/* Constructors and destructors:*/
void RobotArmManager_Init(RobotArmManager* const me);
void RobotArmManager_Cleanup(RobotArmManager* const me);
/* Operations */
void RobotArmManager_computeTrajectory(RobotArmManager* const me, int x, int y, int z, int t);
int RobotArmManager_executeStep(RobotArmManager* const me);
int RobotArmManager_graspAt(RobotArmManager* const me, int x, int y, int z, int t);
int RobotArmManager_zero(RobotArmManager* const me);
struct GraspingManipulator*
RobotArmManager_getItsGraspingManipulator(const RobotArmManager* const me);
```

Source Code: Mediator (Action.h)

```
#ifndef Action_H
#define Action_H
/* class Action */
typedef struct Action Action;
struct Action {
    int manipulatorForce;           /* attribute manipulatorForce */
    int manipulatorOpen;            /* attribute manipulatorOpen */
    int rotatingArmJoint1;          /* attribute rotatingArmJoint1 */
    int rotatingArmJoint2;          /* attribute rotatingArmJoint2 */
    int rotatingArmJoint3;          /* attribute rotatingArmJoint3 */
    int rotatingArmJoint4;          /* attribute rotatingArmJoint4 */
    int slidingArmJoint1;           /* attribute slidingArmJoint1 */
    int slidingArmJoint2;           /* attribute slidingArmJoint2 */
};

/* Constructors and destructors */
void Action_Init(Action* const me);
void Action_Cleanup(Action* const me);
Action * Action_Create(void);
void Action_Destroy(Action* const me);

#endif
```

Source Code: Mediator (RobotArmManager.c graspAt())

```
/* operation graspAt(x,y,z,t) is the main function called by clients of the RobotArmManager. This operation:
1.    zeros the servos
2.    2. computes the trajectory with a call to computeTrajectory()
3. executes each step in the constructed action list
*/
int RobotArmManager_graspAt(RobotArmManager* const me, int x, int y, int z, int t) {
    me->currentStep = 1;
    me->nSteps = 0;
    RobotArmManager_zero(me);
    RobotArmManager_computeTrajectory(me,x,y,z,t); /* updates nSteps and list of actions */
    if ( me->nSteps == 0 ) {
        me->status = -1;
        }
    else {
        do {
            me->currentStep++;
            me->status = RobotArmManager_executeStep(me);
            }
        while (me->status == 0 && me->currentStep < me->nSteps);
        }
    return me->status;
} /* end graspAt() */
```

Source Code: Mediator (RobotArmManager.c executeStep())

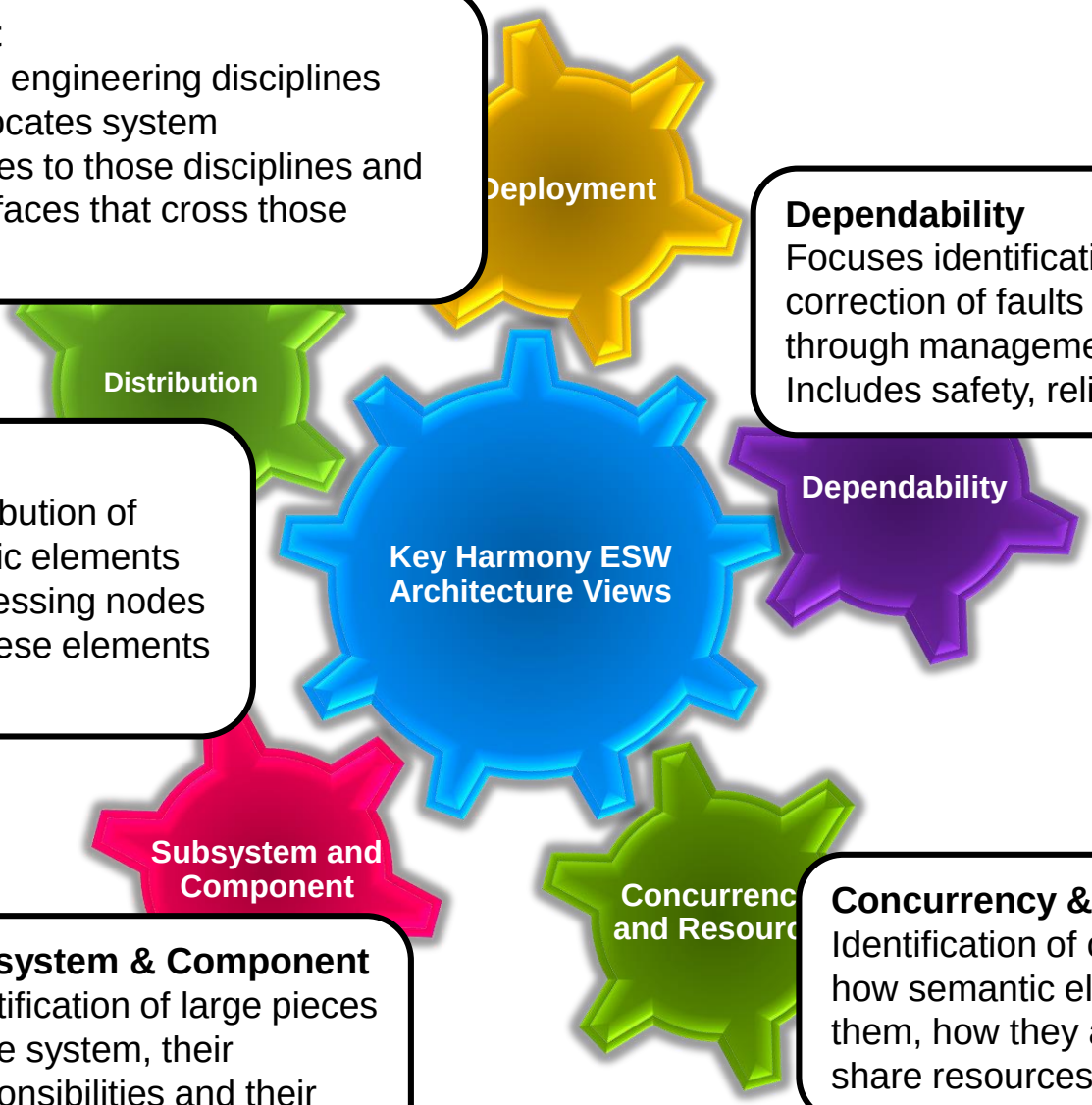
```
#include "RobotArmManager.h"

/* other functions omitted */

/* operation executeStep() This operation executes a single step in the chain of actions by executing all of the
commands within the current action */
int RobotArmManager_executeStep(RobotArmManager* const me) {
  int actionValue = 0;
  int step = me->currentStep;
  int status = 0;
  if (me->itsAction[step]) {
    actionValue = me->itsAction[step]->rotatingArmJoint1;
    status = RotatingArmJoint_rotate(me->itsRotatingArmJoint[0],actionValue);
    if (status) return status;
    actionValue = me->itsAction[step]->rotatingArmJoint2;
    status = RotatingArmJoint_rotate(me->itsRotatingArmJoint[1],actionValue);
    if (status) return status;
    actionValue = me->itsAction[step]->rotatingArmJoint3;
    status = RotatingArmJoint_rotate(me->itsRotatingArmJoint[2],actionValue);
    if (status) return status;
    actionValue = me->itsAction[step]->rotatingArmJoint4;
    status = RotatingArmJoint_rotate(me->itsRotatingArmJoint[3],actionValue);
    status = GraspingManipulator_setMaxForce(me->itsGraspingManipulator, actionValue);
```

Source Code: Mediator (RobotArmManager.c executeStep())

```
if (status) return status;
actionValue = me->itsAction[step]->slidingArmJoint1;
status = SlidingArmJoint_setLength(me->itsSlidingArmJoint[0],actionValue);
if (status) return status;
actionValue = me->itsAction[step]->rotatingArmJoint2;
status = SlidingArmJoint_setLength(me->itsSlidingArmJoint[0],actionValue);
if (status) return status;
actionValue = me->itsAction[step]->manipulatorForce;
if (status) return status;
if (me->itsAction[step]->manipulatorOpen)
    status = GraspingManipulator_open(me->itsGraspingManipulator);
else
    status = GraspingManipulator_close(me->itsGraspingManipulator);
};
return status;
} /* end executeStep() */
```

Five Key Views of Architecture

Deployment
Identifies the engineering disciplines involved, allocates system responsibilities to those disciplines and defines interfaces that cross those boundaries



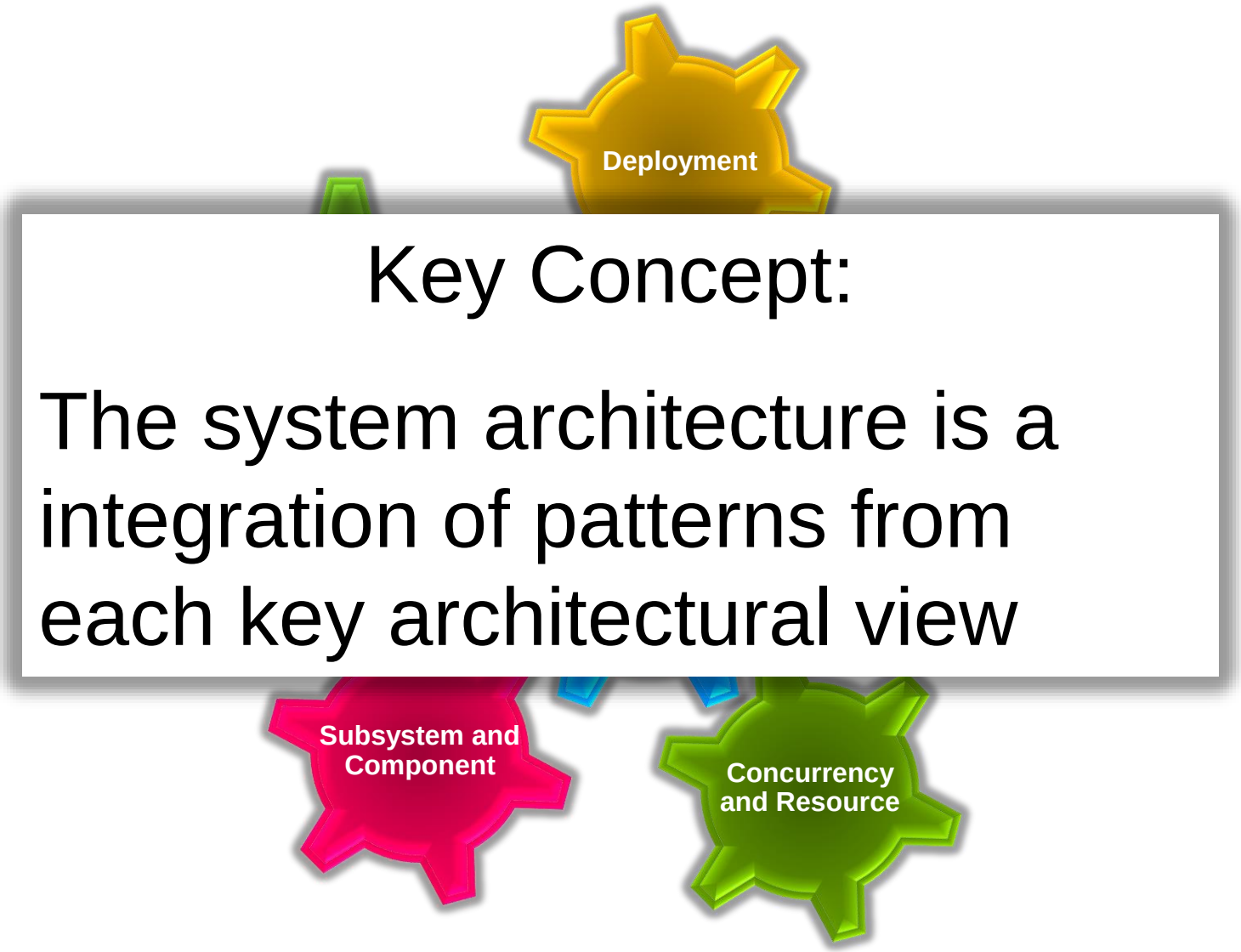
Dependability
Focuses identification, isolation and correction of faults as the system runs through management of redundancy. Includes safety, reliability, & security

Distribution
Focuses on the distribution of services and semantic elements across different processing nodes and identifies how these elements collaborate

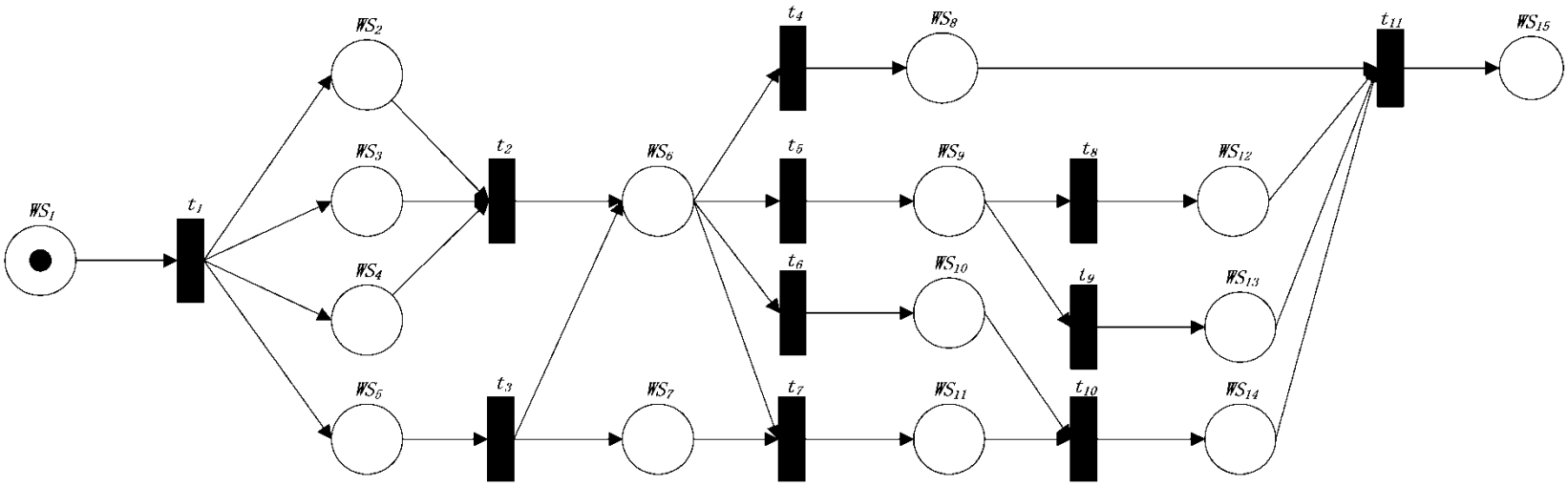
Subsystem & Component
Identification of large pieces of the system, their responsibilities and their interfaces

Concurrency & Resource
Identification of concurrency units, how semantic elements map to them, how they are scheduled & share resources

Five Key Views of Architecture



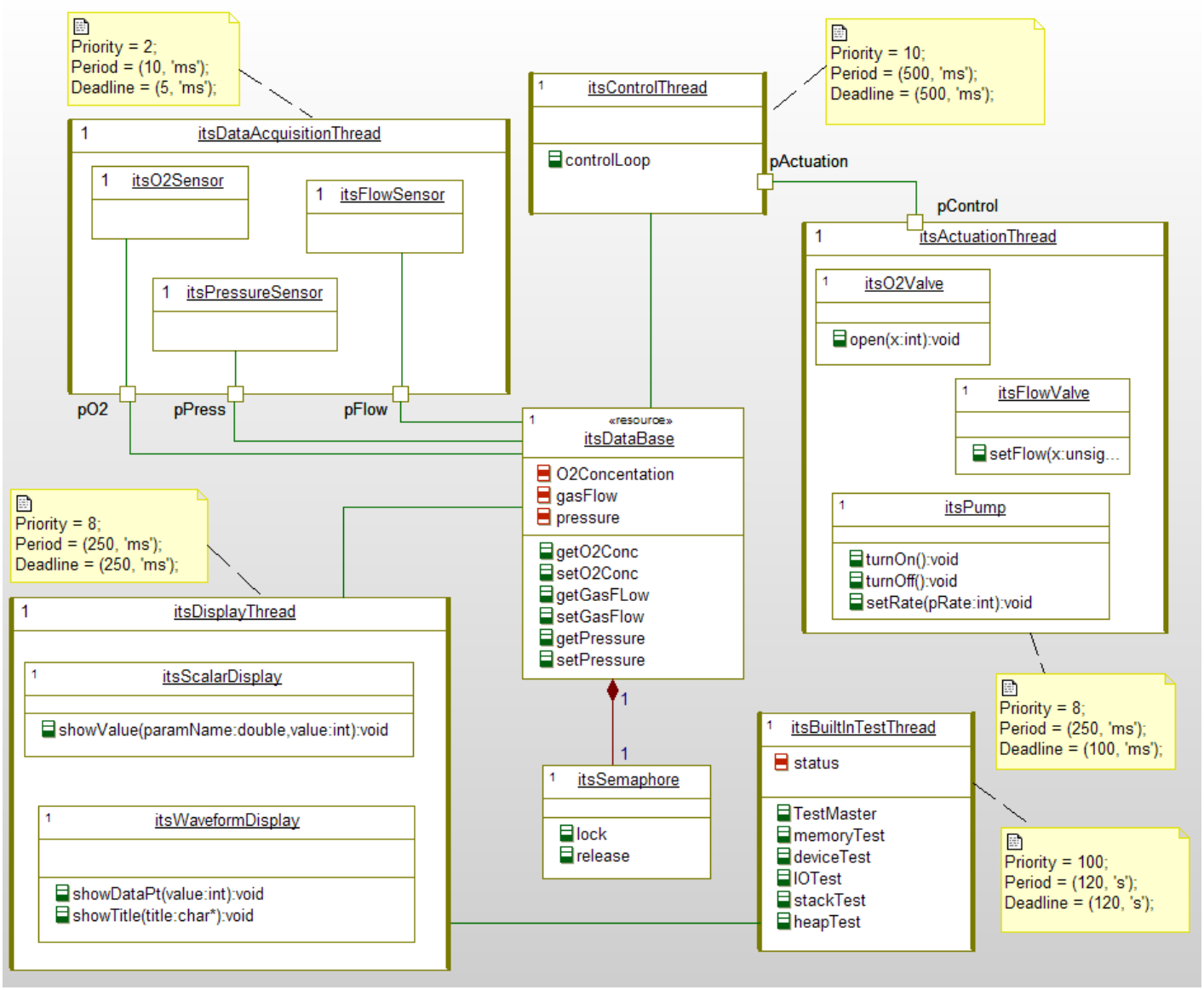
Concurrency Architecture Patterns



Task Diagram

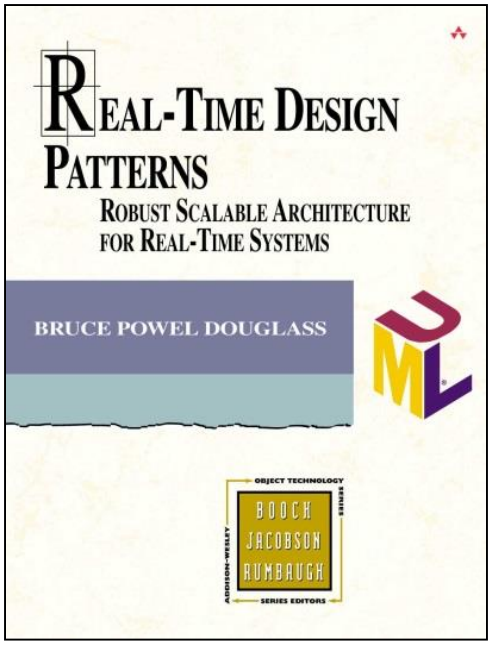
- A task diagram is a class diagram that shows only model elements related to the concurrency model
  - Active objects
  - Semaphore objects
  - Message and data queues
  - Constraints and tagged values
- May use opaque or transparent interfaces

Task Diagram



Task Scheduling Patterns

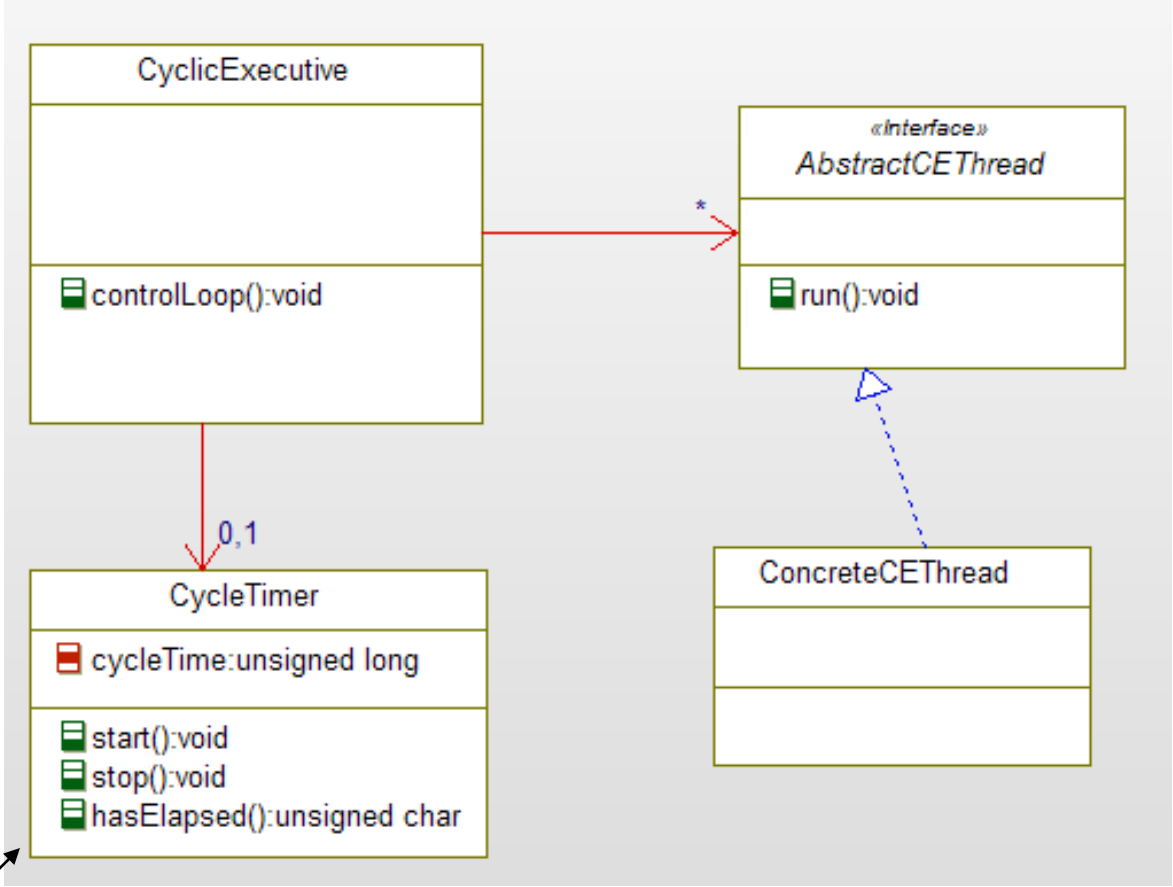
- **Priority-based preemptive**
  - Highest priority task not blocked runs preferentially
  - May be static (priority assigned at design) or dynamic (priority assigned at run-time)
- Non-preemptive
  - Round robin executes tasks in turn
  - May require “cooperative multitasking”
- Time Driven Multiplexed Architecture (TDMA)
  - Each task is given a specific time-slice in a round-robin fashion
- **Cyclic executive**
  - Run a set sequence in a particular order
  - Each task runs to completion
- **Interrupt**
  - No scheduling per se, just a set of interrupt handlers
  - Requires that handlers are short (relative to arrival frequency) and atomic



Cyclic Executive Pattern

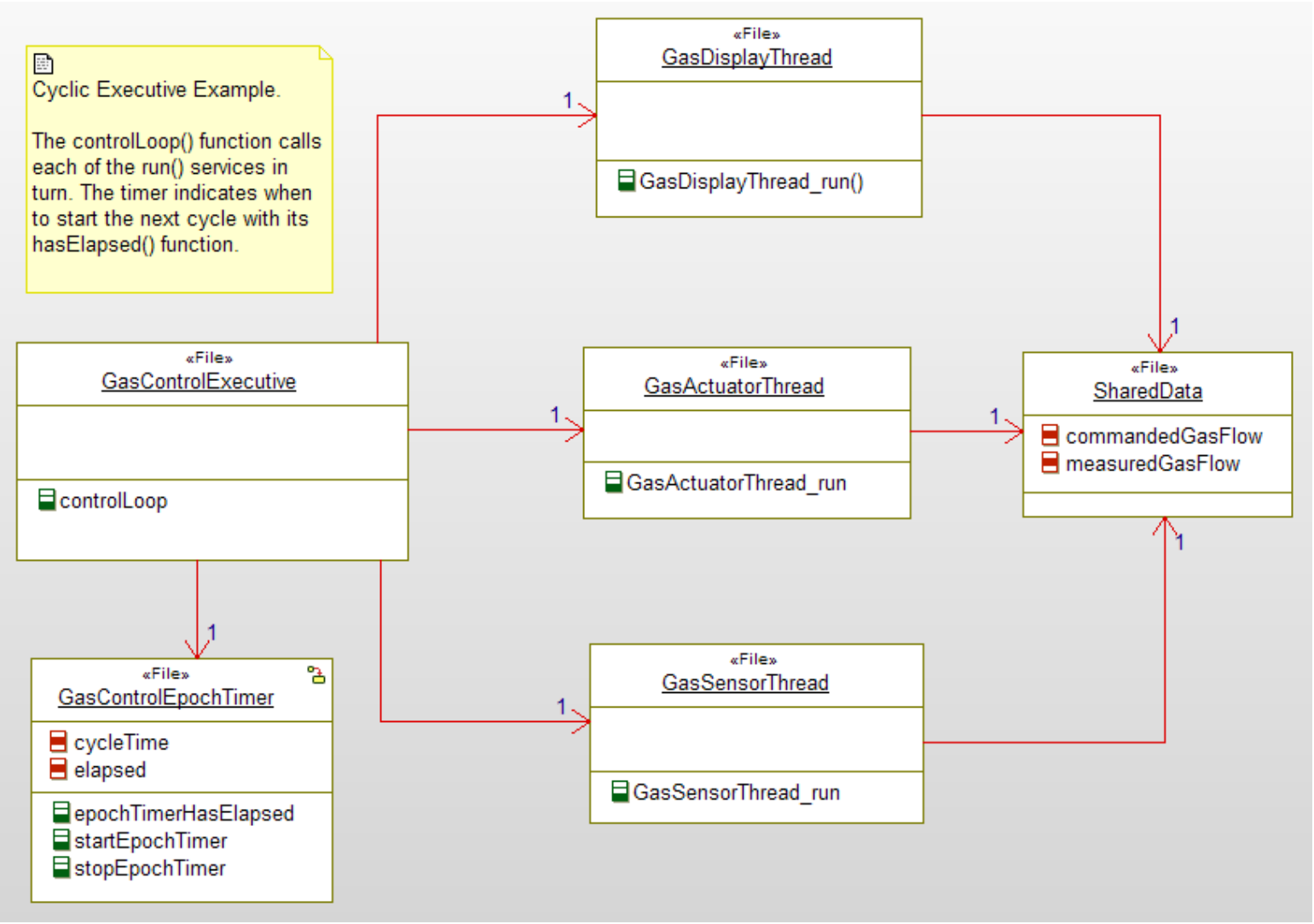
- Problem
  - Need for a simple means to execute a known, finite set of task, or
  - You want to simplify safety certification
- Solution
  - Have each task run-to-completion
  - Use an executive to order the task execution
- Consequences
  - Very simple implementation
  - Demonstrably suboptimal in terms of time to respond to incoming events
  - Applies best to simple task sets that run to completion
  - May require tuning

Cyclic Executive Pattern



This timer is only needed for the Timed Cyclic Executive variant which starts the cycle on a specific time period

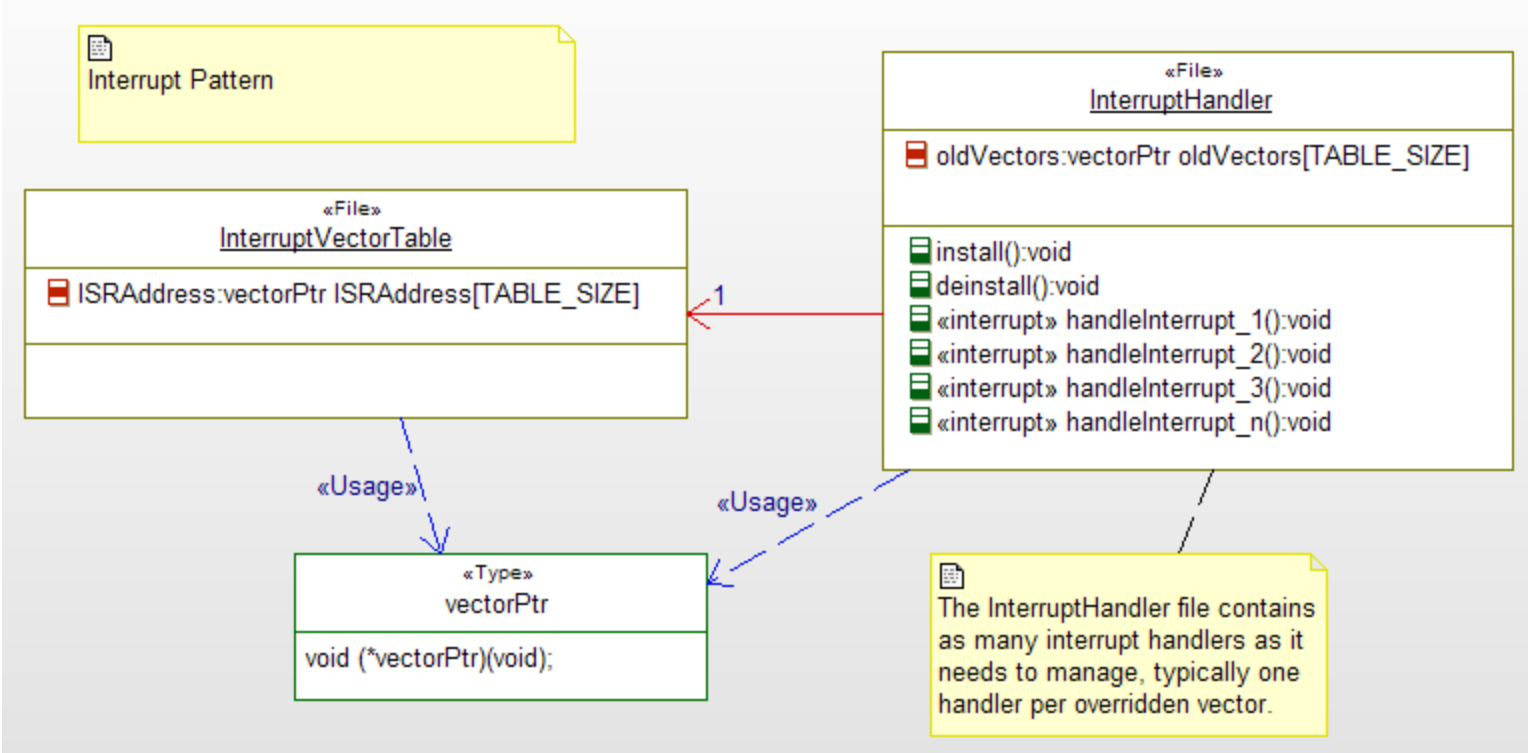
Cyclic Executive Pattern Example



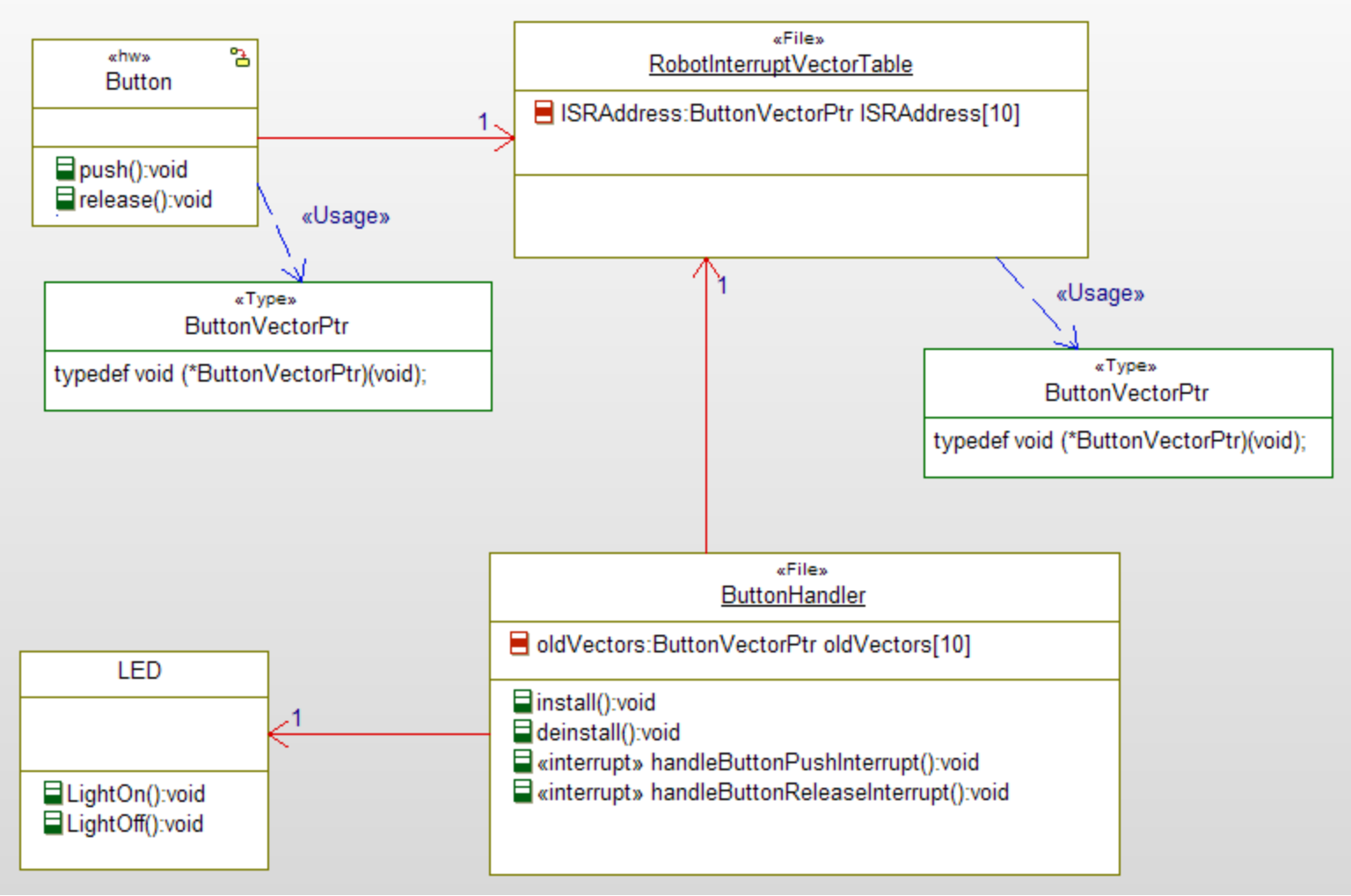
Interrupt Pattern

- Problem
  - You want to process events from external sources, including hardware
- Applicability
  - Events come at non-predictable intervals and polling would be ineffective
- Solution
  - Write a device driver that in its constructor inserts the address of an interrupt function into the CPU interrupt vector table and in its destructor restores the previous vector
  - Care must be taken in C++ (use static functions)
- Consequences
  - Easy implementation
  - Highly responsive to incoming events
  - Interrupt handles must be short or high frequency events may be lost
  - Care must be taken to avoid deadlock when ISR updates a protected shared resource

Interrupt Pattern



Interrupt Pattern Example



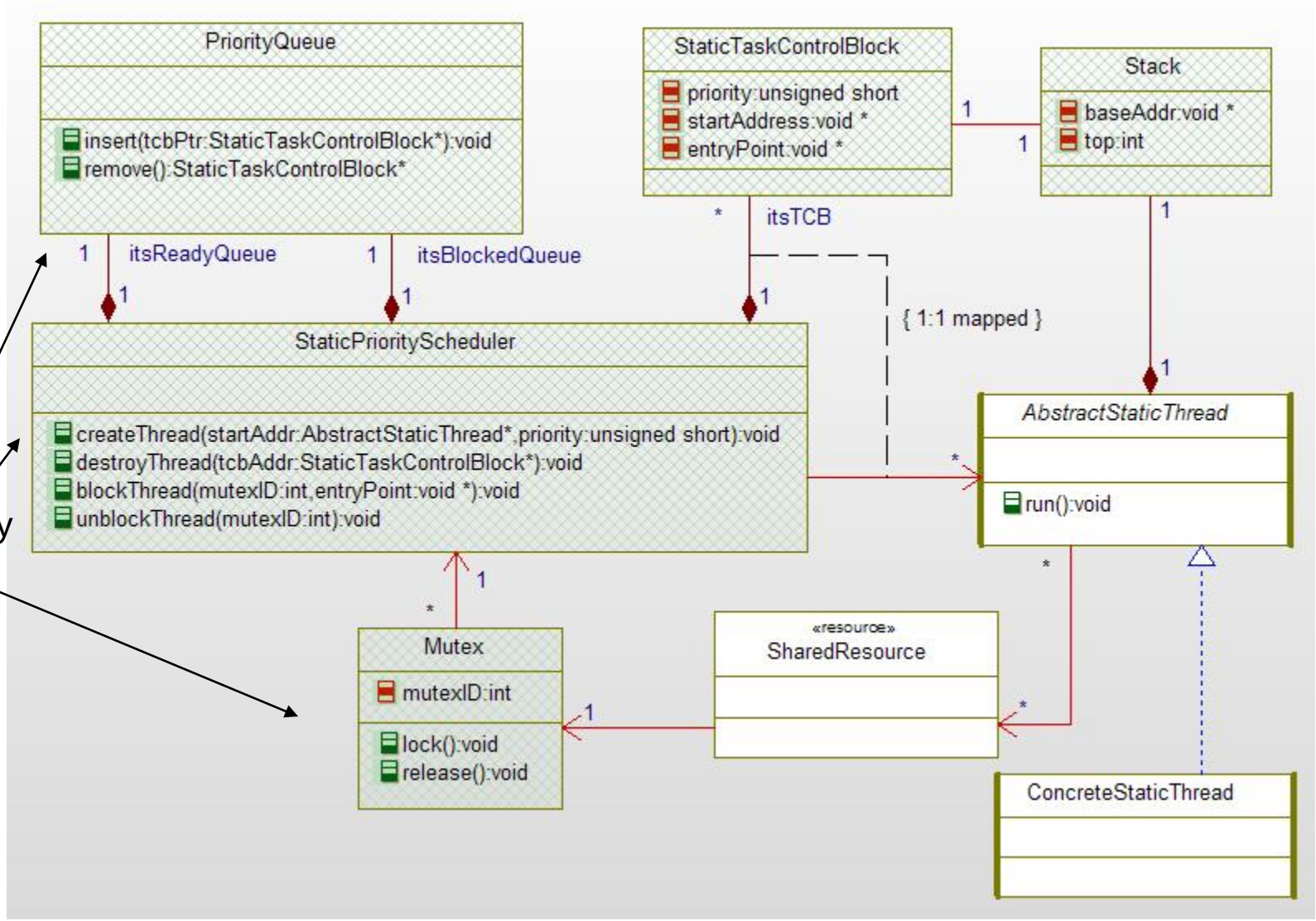
Static Priority Pattern

- Problem
  - In a multitasking environment, need a set of rules to govern how ready-to-run threads are scheduled
- Solution
  - Use a static priority for each task and implement the rule that the highest priority task ready to run preempts and runs preferentially

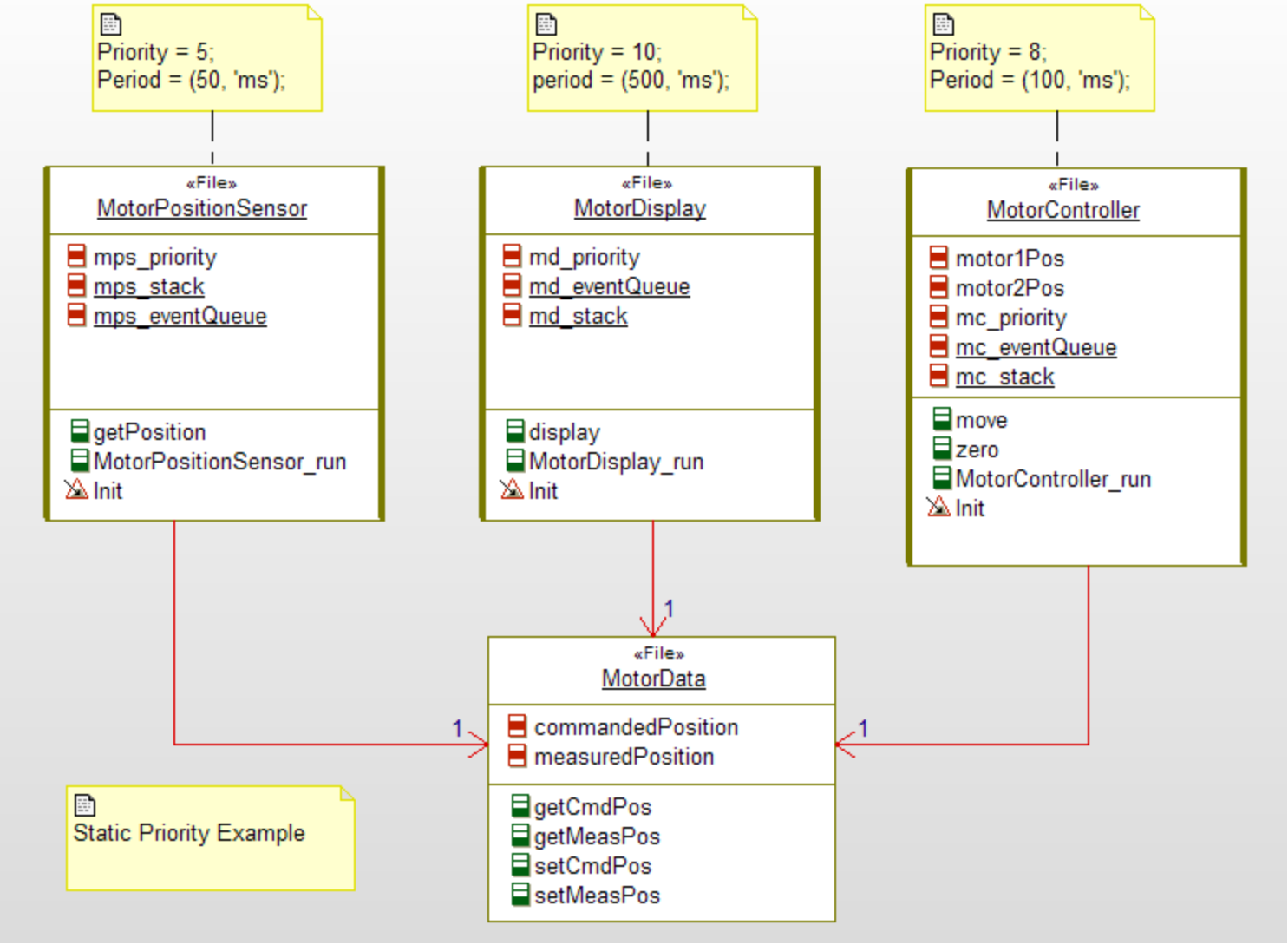
Static Priority Patten

- Consequences
  - Supported by many RTOSes
  - Can be statically analyzed for schedulability (e.g. RMA analysis)
  - Easy to annotate UML thread model with the appropriate properties
    - Priority
    - Worst Case execution time
    - Blocking time, etc.
  - Scales to many threads well
  - RMS is the most common instantiation
    - RMS is OPTIMAL
    - RMS is STABLE
  - Naïve implementation can lead to unbounded priority inversion

Static Priority Pattern

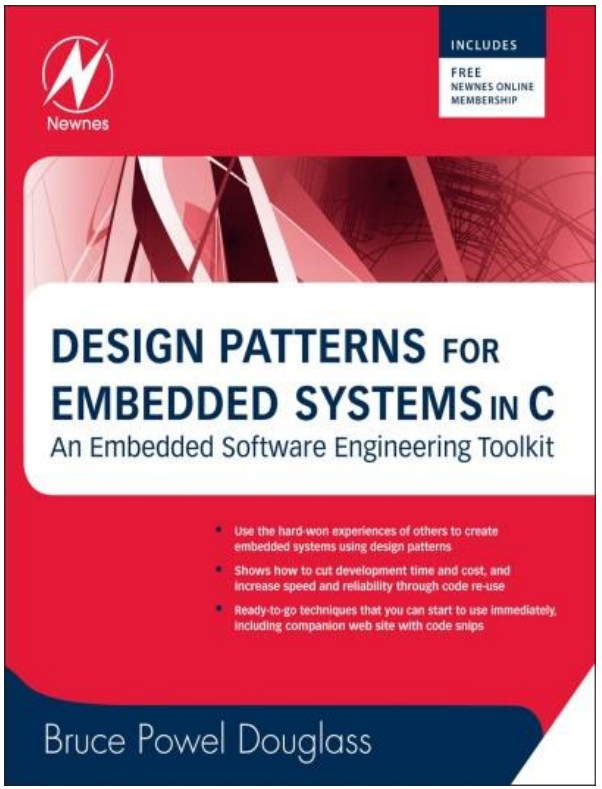


Static Priority Example



Safety and Reliability Patterns

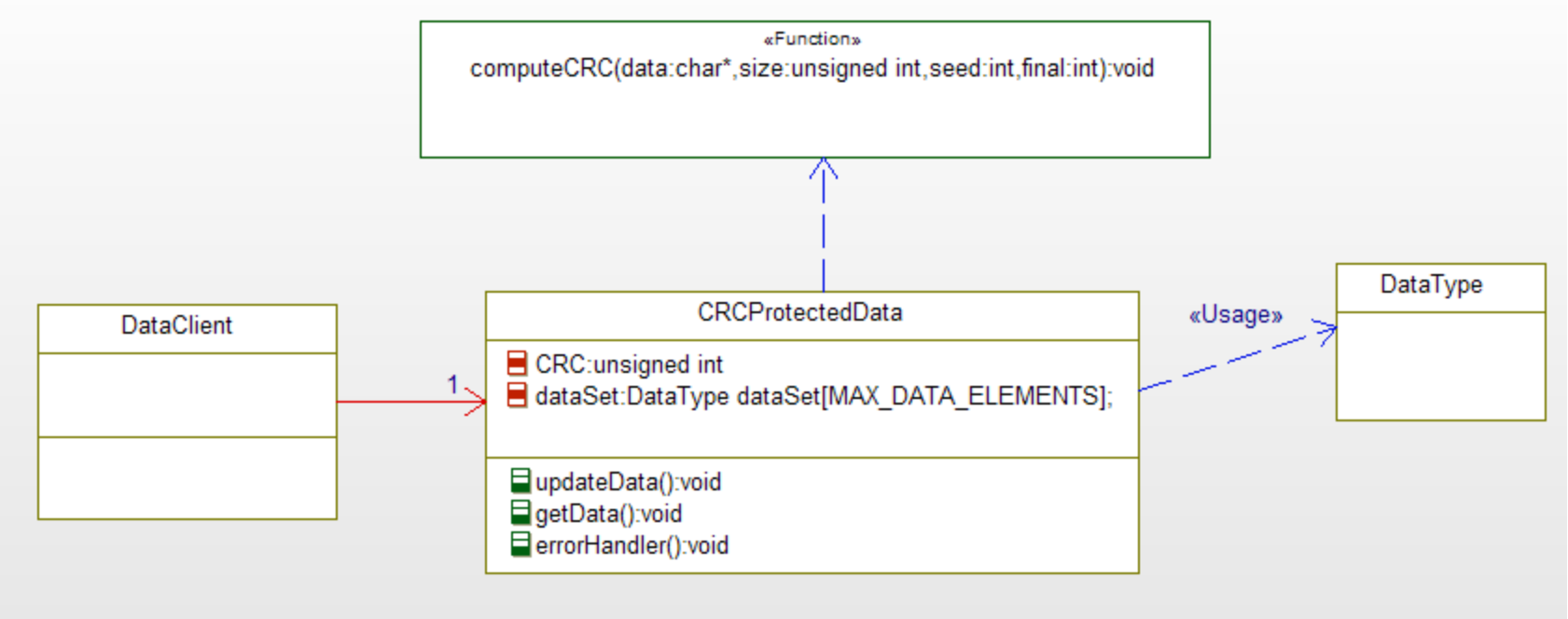
- Isolation Pattern
- **CRC Pattern**
- **Smart Data Pattern**
- Protected Single Channel Pattern
- Homogeneous Redundancy Pattern
- Heterogeneous Redundancy Pattern
- Triple Modular Redundancy Pattern
- Monitor-Actuator Pattern



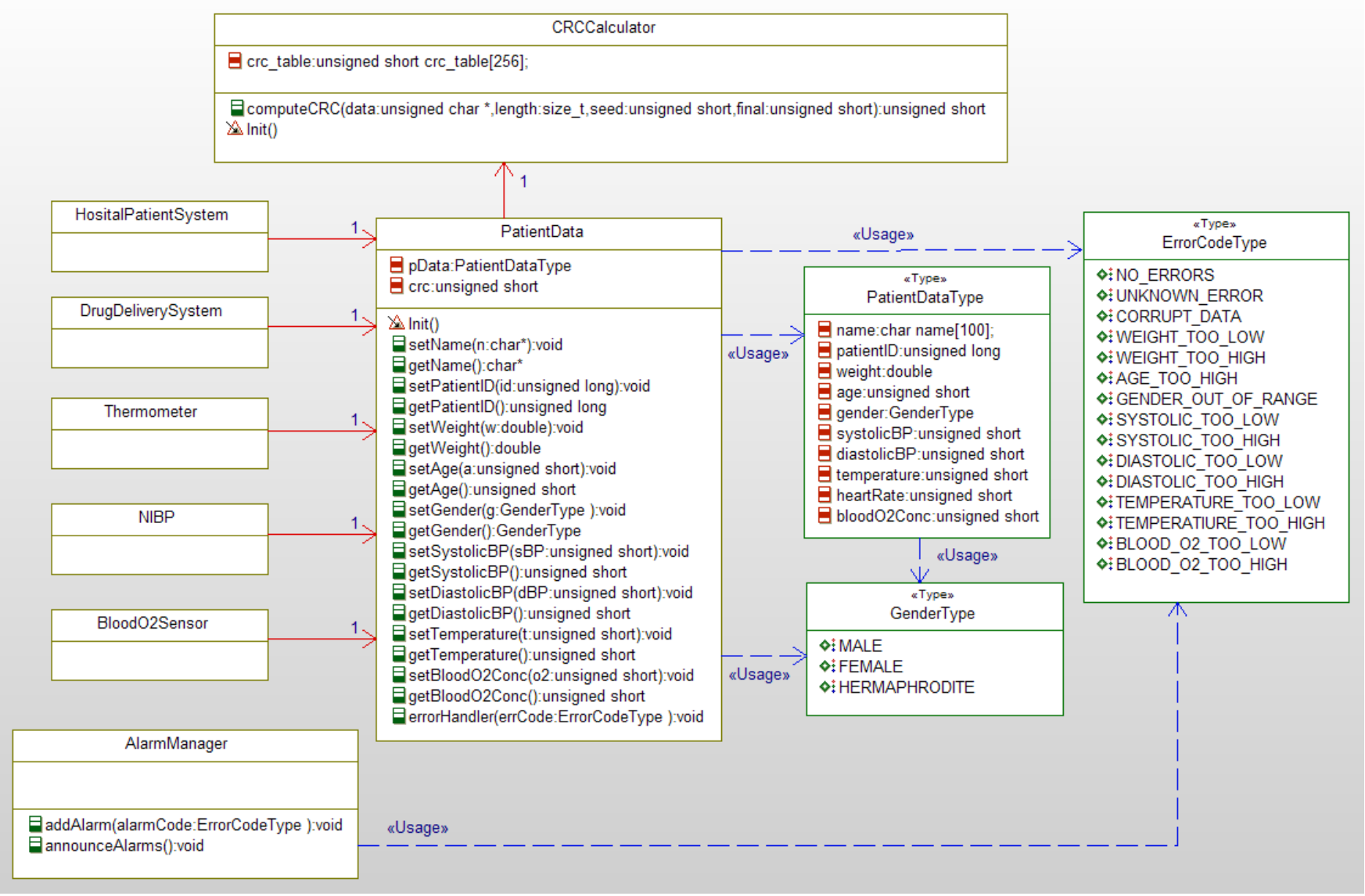
CRC Pattern

- Problem
  - This pattern addresses the problem that variables may be corrupted from a variety of causes such as environmental factors (such as EMI, heat, and radiation), hardware faults (such as power fluctuation, memory cell faults, and address line shorts), or software faults (other software erroneously modifying memory). This pattern addresses the problem of data corruption in large data sets.
- Solution
  - The pattern adds cyclic redundancy checks to identify data corruption and trigger appropriate action when it occurs
- Consequences
  - CRC uses a small amount of memory for strong bit-corruption identification
  - Table-driven implementations use additional block of memory to hold table but are computationally efficient

CRC Pattern



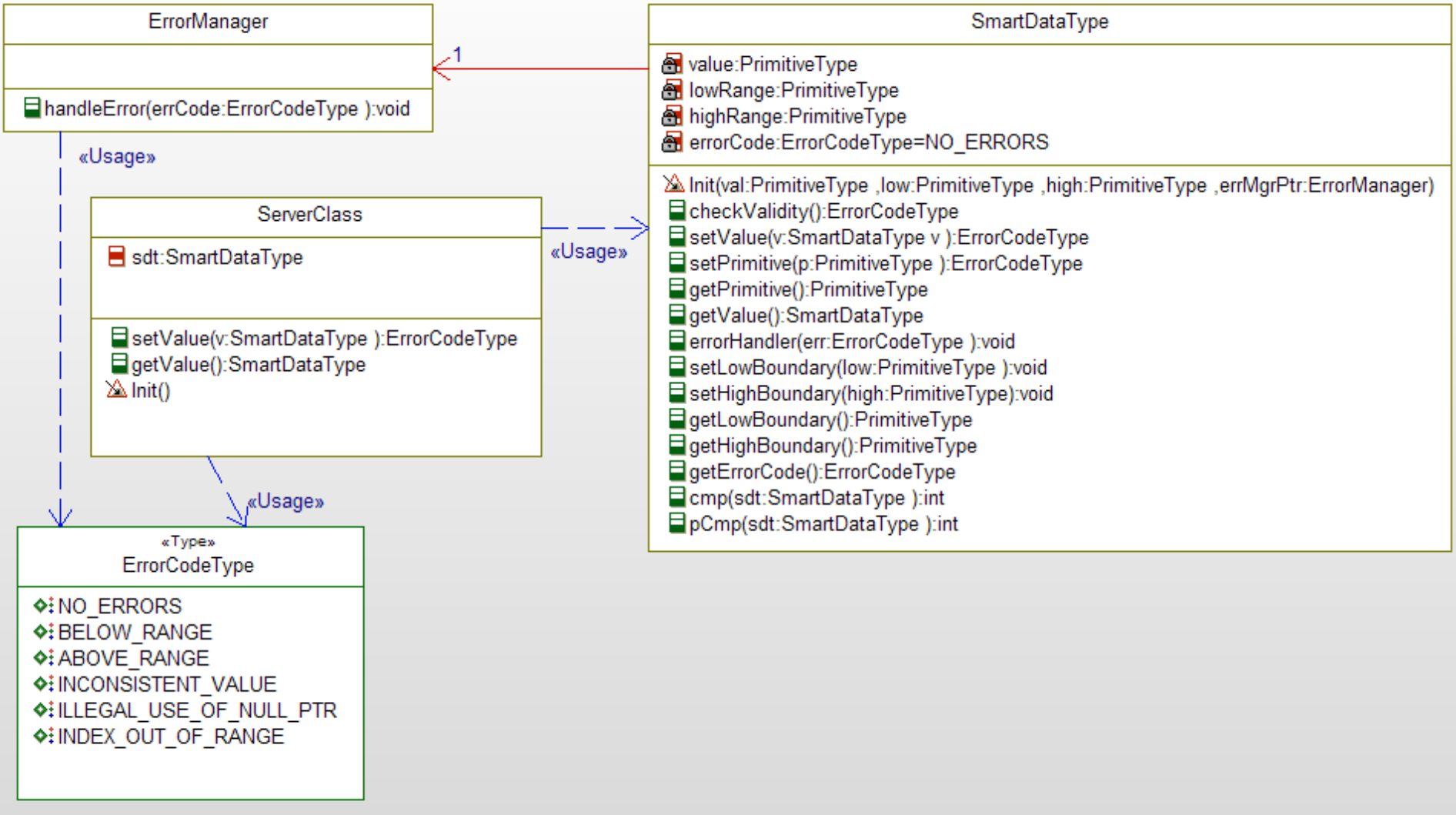
CRC Pattern Example



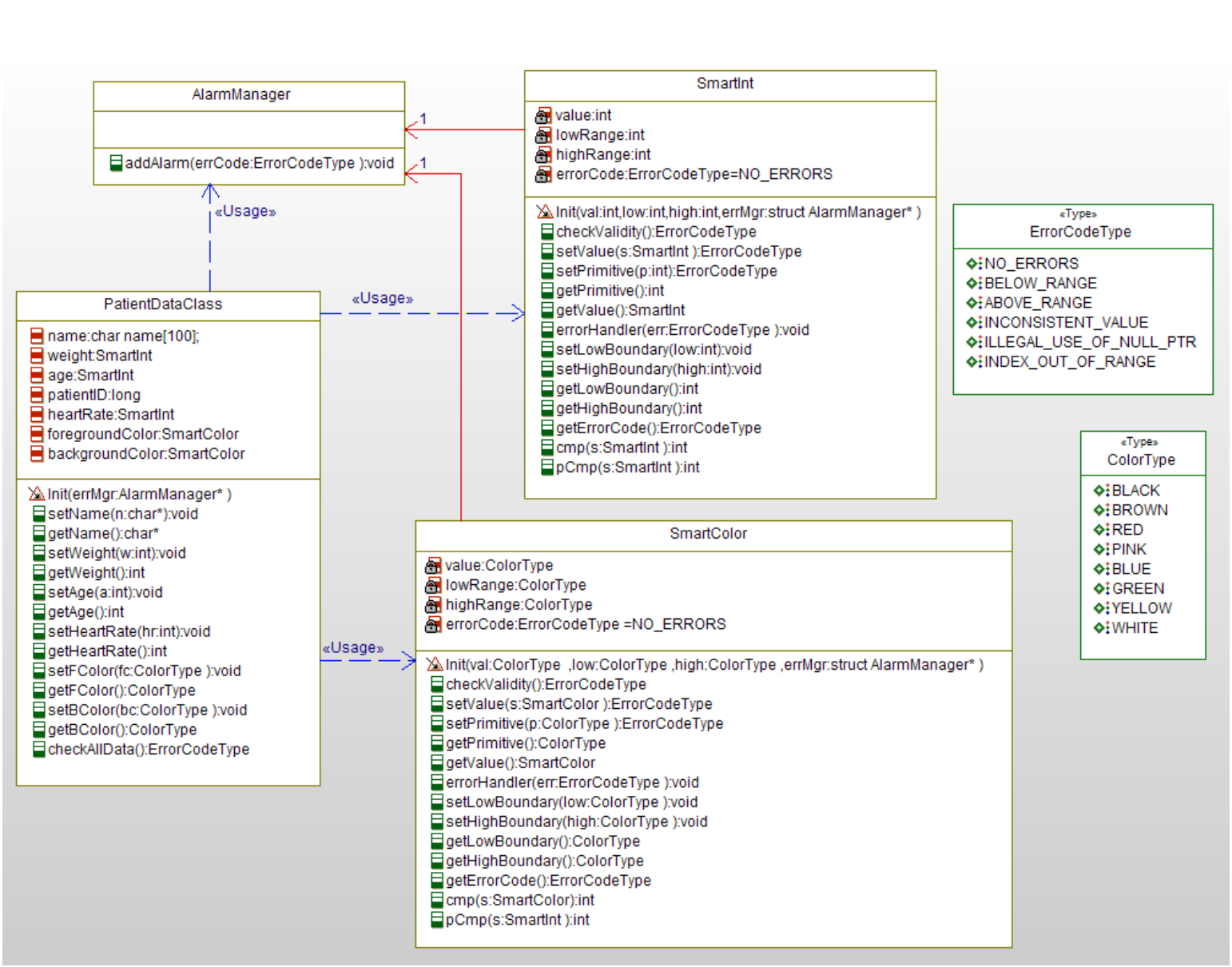
Smart Data Pattern

- Problem
  - The problem this pattern addresses is to build functions and data types that essentially check themselves and provide error detection means that cannot be easily ignored.
- Solution
  - The key concepts of the pattern are to
    - Build self-checking types whenever possible
    - Check incoming parameter values for appropriate range checking
    - Check consistency and reasonableness among one or a set of parameters.
- Consequences
  - The downside for using smart data types is the performance overhead for executing the operations.
  - The upside is that the data is self-protecting and provides automatic checking when the data is set.
  - It is also possible for the programmers to avoid using the functions and access the values directly if they are so inclined, defeating the purpose of the smart data type.

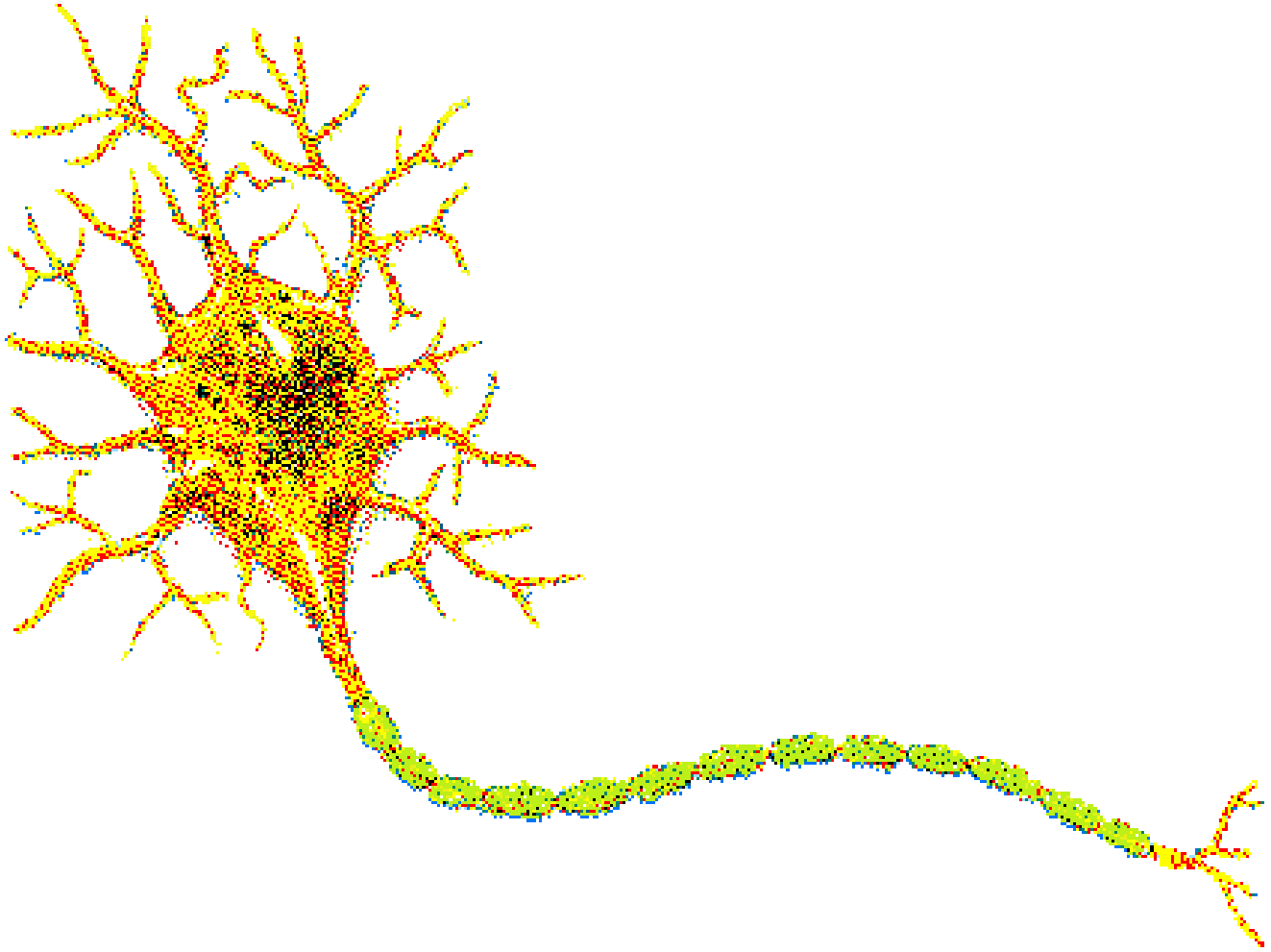
Smart Data Pattern



Smart Data Pattern Example

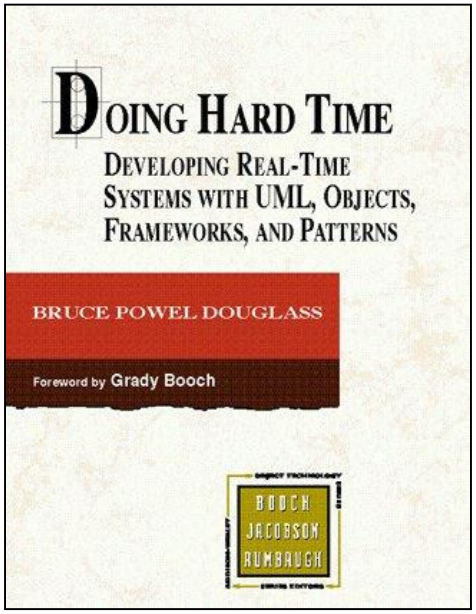


State Behavioral Patterns



State Patterns

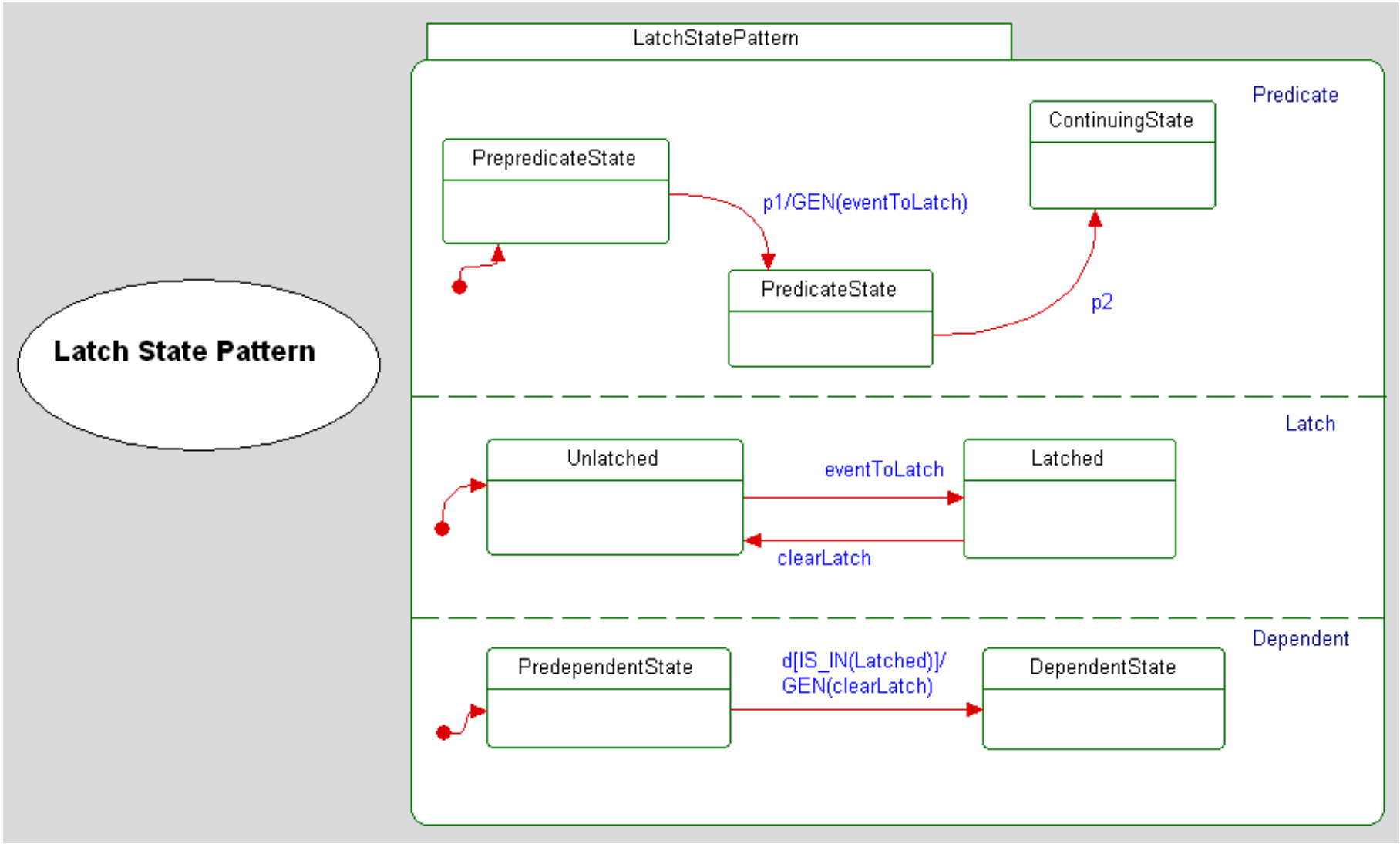
- **Latch State Pattern**
- Polling State Pattern
- Queued Data State Pattern
- Any State Pattern
- **Transaction State Pattern**
- Counting Barrier State Pattern
- Random State Pattern
- Null State Pattern
- Watchdog State Pattern
- Retriggerable Counter State Pattern



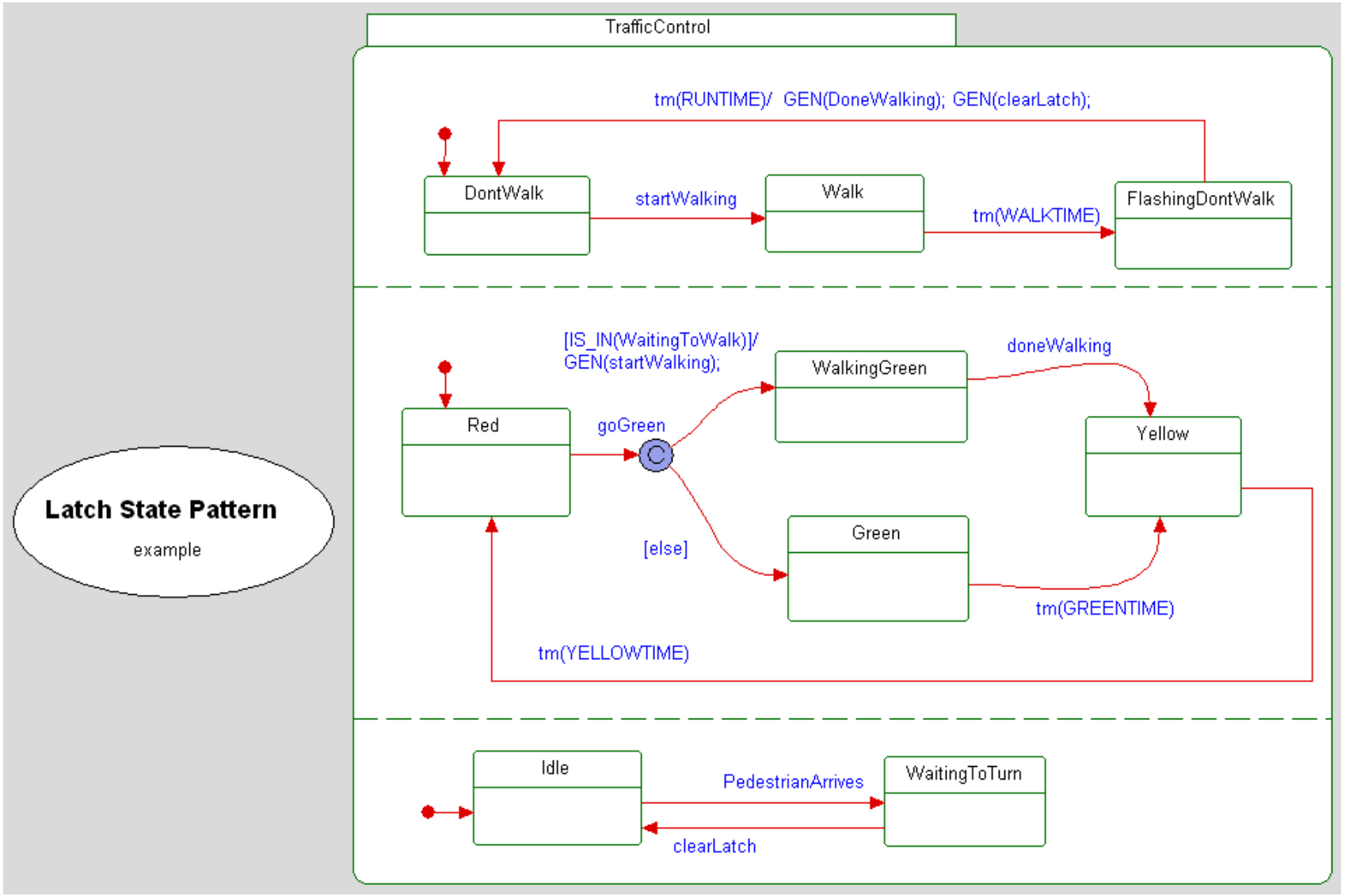
Latch State Pattern

- Problem
  - You want to remember that an event has occurred so that you can process it later
- Applicability
  - When the event can come at any time, but the object may not be able to react to it except at specific points
  - When you want to remember that a state has been visited and use that information later
- Solution
  - Create a “latch” state to remember the arrival of the event of interest and clear the latch when the object has consumed it
- Consequences
  - It is a lightweight means to synchronize to independent behaviors when a latching condition is required.
  - Other kinds of latches may be constructed. For example, an inhibitory latch (a latch which, when active, inhibits the progress of an independent activity) can be easily constructed by applying a NOT operator (e.g. [!IN(Latched))]) within the guard condition.

Latch State Pattern



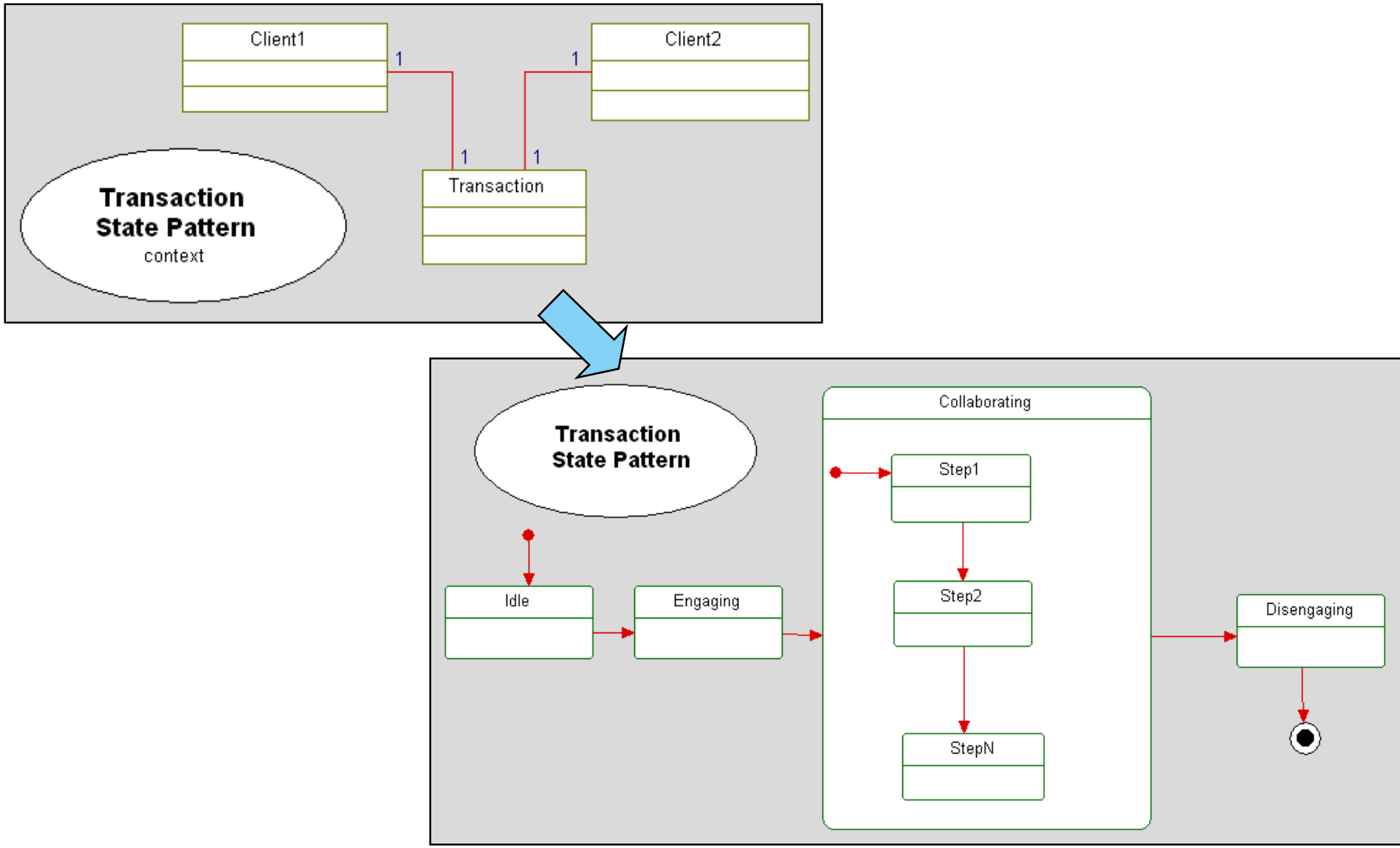
Latch State Pattern Example



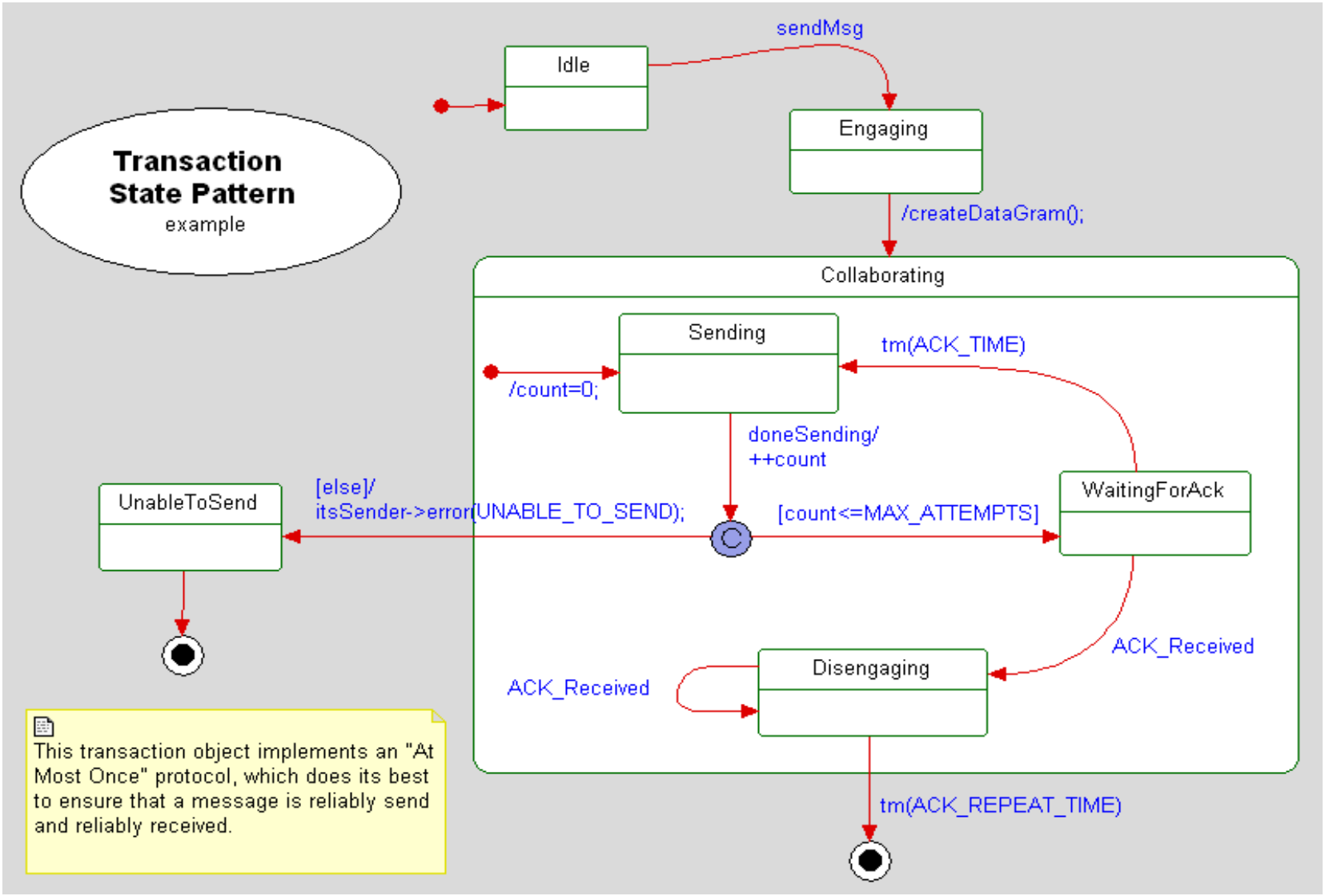
Transaction State Pattern

- Problem
  - You want to have an agent mediate the interaction of two objects
- Applicability
  - The interaction between two objects progresses in a series of steps (states)
  - You want to manage multiple such interactions simultaneously
- Solution
  - Reify the interaction as a separate objects and specify the steps of their interaction as states in the transaction object
- Consequences
  - A very flexible means for managing complex interactions
  - Can be extended to support multiple clients (see the architectural Rendezvous pattern)

Transaction State Pattern



Transaction State Pattern Example



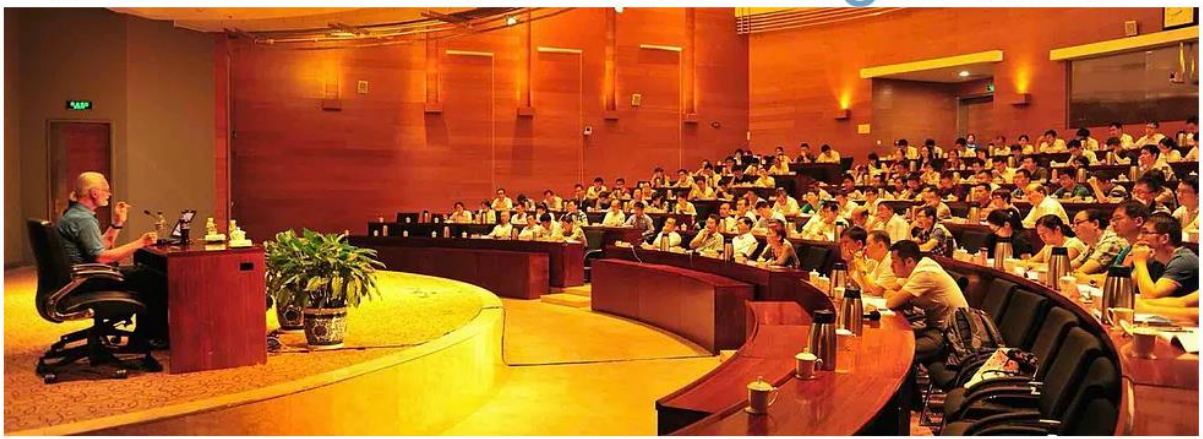
Bruce Powel Douglass, Ph.D.

Log In

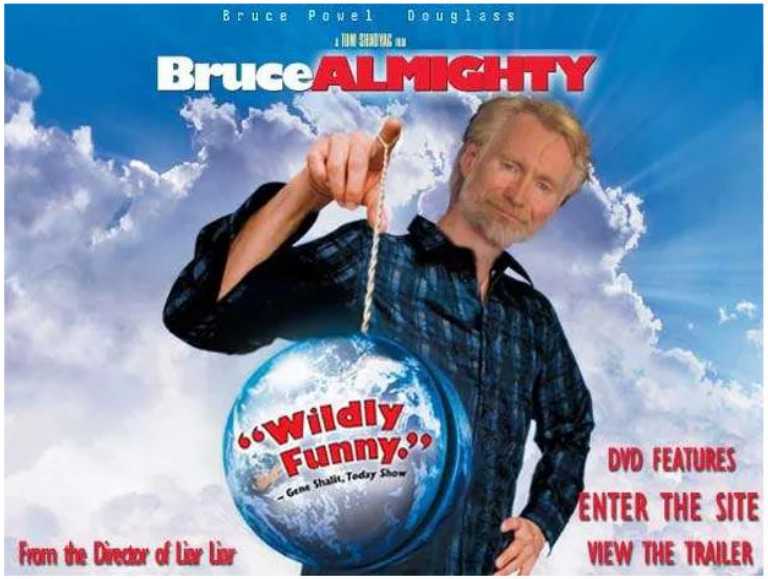
Content | Services | Public Interest | Blog | What's New | Forum | About | Comments | Site Map | Geekosphere | Member

Real-Time Agile Systems and Software Development

Welcome to www.bruce-douglass.com



Site Map



You've found yourself on **www.bruce-douglass.com,** my web site on all things real-time and embedded.

On this site you will find papers, presentations, models, forums for questions / discussions, and links (lots of links) to areas of interest, such as

- Developing Embedded Software
- Model-Driven Development for Real-Time Systems
- Model-Based Systems Engineering
- Safety Analysis and Design
- Agile Methods for Embedded Software
- Agile Methods for Systems Engineering
- The Harmony agile Model-Based Systems Engineering process
- The Harmony agile Embedded Software Development process
- Models and profiles I've developed and authored
- List and links to many of my books.