ZZZ Software Architecture

Document Number:
ZZZ-DSGN-45
Version: 1.1
10/2/15 12:53 PM

ZZZ Company

REVISION LOG

Version	Description of Change(s)	Initials	Date
1.0	First Draft	AP	17 Sep 97
1.1	Inspection revisions	AP	19 Feb 98

Table Of Contents

List Of Figures

1. INTRODUCTION

This document describes the following aspects of the Control System Initiative (ZZZ) software architecture:

- principal classes
- inter-agent protocols
- handling failures and exceptions
- interface to safety system

To facilitate maintenance and reuse, we specify an object-oriented software architecture for the ZZZ. The particular OO design methodology we employ is *Responsibility-Driven Design* (*RDD*) [Wirfs-Brock, 1990]. The starting point for this methodology is to identify the principal classes of a software system. This is typically done by identifying the physical and conceptual objects that are significant to users of the system and that require representation in software. *Responsibilities* or services are then assigned to the principal classes. The analytical phase of RDD concludes with the identification of *collaborations* between the principal classes. Subsequent phases refine and augment the initial classes and their relationships and specify detailed class interfaces.

2. APPLICABLE DOCUMENTS

2.1 ZZZ Documents

ZZZ HSMC Baseline ("Red Cover") Control System Specifications, 6 January 1997.

2.2 Non-ZZZ Documents

[Wirfs-Brock, 1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

3. PRINCIPAL CLASSES

3.1 Responsibilities

The requirements for the ZZZ control software include the following responsibilities:

R1. Checking and configuring the system
R2. Initiating system operation and monitoring system status
R3. Transporting cells to and between process stations (PSs) by using continous motion conveyors
(CMCs) and continous feed indexers (CFIs)

R4. Managing the global conveyor system to maximize throughput and minimize dancer-limit failures
R5. Notifying PSs about the status of oncoming cells
R6. Actuating PSs according to work specifications, cell status, and conveyor status
R7. Recording information about the processing of each cell
R8. Effecting controlled system shutdown when required
R9. Handing subsystem failures either by restoring the subsystem, replacing it, or shutting down the overall system
R10. Resuming operation after either controlled or uncontrolled shutdown
R11. Providing an operator interface that supports system operation and problem diagnosis/resolution

The principal physical objects in the ZZZ system include:

A. Process stations
B. Continuous motion conveyors
C. Continuous feed indexers
D. The safety system

Since each of these objects must be controlled and/or monitored by the ZZZ control software, it is natural to define object classes to encapsulate their associated data and behavior. The responsibilities of the software can then be allocated as follows. Responsibility R3 is shared between class *ContinuousMotionConveyor* and class *ContinuousFeedIndexer*. Responsibility R6 is shared between class *ProcessStation* and class *ContinuousFeedIndexer*. The remaining responsibilities cannot be allocated solely to classes modelling physical objects in the ZZZ system. Note that responsibilities R1, R2, R4, R5, R8, R9 and R10 involve the coordination of system components.[1] At this stage in the design we choose to allocate responsibilities R1, R2, R4, R5, R7, R8, R9, and R10 primarily to a single *Coordinator* object.[2] For the time being, we defer the assignment of responsibility R11.

3.2 Coordinator

In the current design, there is a single *Coordinator* agent, which is primarily responsible for the following:

R1. Checking and configuring the system
R2. Initiating system operation and monitoring system status
R4. Managing the global conveyor system to maximize throughput and minimize dancer-limit failures
R5. Notifying PSs about the status of oncoming cells
R7. Recording information about the processing of each cell
R8. Effecting controlled system shutdown when required
R9. Handing subsystem failures either by restoring the subsystem, replacing it, or shutting down the overall system
R10. Resuming operation after either controlled or uncontrolled shutdown

[1] Cell transport also involves system coordination if cells can be conditionally switched or routed between alternative conveyor elements.
[2] If the interface of the *Coordinator* became excessively complex, it would be desirable to divide these responsibilities between multiple classes.

Responsibilities R1, R2, R4, R8, R9, and R10 imply responsibilities to check, configure, start, interrogate, adjust, and stop subsystems such as CFIs, CMCs, and PSs. The *Coordinator* delegates these responsibilities to the subsystems' controlling agents, e.g., *ContinuousFeedIndexer*, *ContinuousMotionConveyor*, and *ProcessStation* agents.

Responsibilities R5 and R7 involve the exchange of cell-specific information between the *Coordinator* and the *ProcessStation* objects. The *Coordinator* obtains information about the status of cells from the *ProcessStation* agents that process them, and it passes this information on to other *ProcessStation* agents that require it. It is therefore necessary to specify an inter-controller protocol for the communication of cell-specific data between the *Coordinator* and the *ProcessStation* agents. This is one of the subjects of Section 4. The *Coordinator* has responsibility for recording information about the processing history for each cell because it has overall knowledge of the ZZZ manufacturing process. At a minimum, this information should indicate which PSs a cell visited and whether any process steps failed. Additional information might also be included, e.g., the results of testing a cell.

Because the *Coordinator* is responsible for supervisory control of the entire ZZZ system, it must know the overall structure of the system and it must monitor the system's state. It is the *only* agent that should have access to all of this information. Hiding the system's structure from other agents limits coupling between them and reduces the reprogramming entailed by changing the structure.

3.3 ProcessStation

There is a *ProcessStation* agent corresponding to each physical process station on the ZZZ line. This agent is responsible for actuating the PS to process cells according to their work specification and status. There are likely to be separate *ProcessStation* agents for inputting cells, outputting them, applying sealant, inserting separators, inserting KOH, inserting anode gel, inserting collectors, crimping and closing cells, and disposing of defective cells. A *ProcessStation* agent issues work reports describing the results of processing individual cells. These should indicate whether processing was successful or not; they might also contain additional information, such as test results or other measurements. Work reports are sent to the *Coordinator*, in support of its notification and record-keeping responsibilities (R5 and R7). *ProcessStation* agents are likely to execute on computers that are distributed around the ZZZ line. It is possible that as few as one or as many as several *ProcessStation* agents will be allocated to an individual controller.

A *ProcessStation* agent must be capable of configuring, starting, and stopping its PS at the request of the *Coordinator*. A PS may be stopped either in response to a failure or simply because further production is not required. A *ProcessStation* agent monitors its PS's status and provides this to the *Coordinator* in support of the latter's responsibilities to monitor system status and handle component failures.

ProcessStation agents need not and should not embody knowledge of the ZZZ line's overall structure, so that they need not be reprogrammed when the line structure is changed. In particular, *ProcessStation* agents should not be aware of the disposition of other *ProcessStation* agents. The *Coordinator* should mediate interactions between a *ProcessStation* agent and the rest of the system.

3.4 ContinuousFeedIndexer

There is one *ContinuousFeedIndexer* agent for each CFI on the ZZZ line. This agent is responsible for:

- Controlling its CFI and dancer

- Interacting with the *Coordinator* when the system is configured, started or stopped, when a failure occurs, and when the *Coordinator* requests its status
- Monitoring the health of its CFI and informing the *Coordinator* of malfunctions
- Supporting the *Coordinator's* responsibility to manage the global conveyor system
- Synchronizing the operation of its CFI and associated PSs, by interacting with the latter's *ProcessStation* agents

3.5 ContinuousMotionConveyor

The *ContinuousMotionConveyor* agent naturally controls the continuous motion conveyor. It is also responsible for:

- Interacting with the *Coordinator* when the system is configured, started or stopped, when a failure occurs, and when the *Coordinator* requests its status
- Monitoring the health of its CMC and informing the *Coordinator* of malfunctions
- Supporting the *Coordinator's* responsibility to manage the global conveyor system

3.6 Remote Proxies

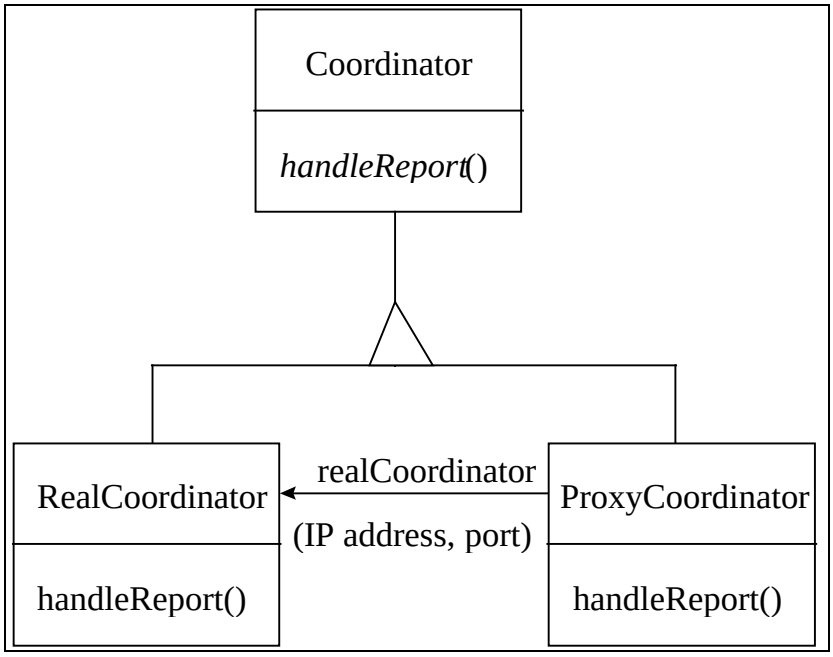

Figure 1: Coordinator and Proxy Classes

In cases where communicating agents like the *Coordinator* and *ProcessStation* agents are physically distributed, **remote proxies** will be used to encapsulate the details of network communication and to make distribution of objects transparent to their clients. A remote proxy is a local stand-in for a remote server and provides exactly the same interface. Typically, the proxy and the real server are derived from the same abstract class (see
Figure 1). The implementation of a proxy method marshalls the method parameters for network transmission and employs transport services, such as TCP sockets, to encapsulate the method call as a

message which is sent to the host of the real server. The message is handled by a server adapter which unmarshals the parameters and invokes the appropriate method of the real server. The server adapter also passes back any return value or exception that is thrown by the real method. Note that when remote proxies are used, the way network communication is implemented can be changed without affecting clients.[3] In fact, clients will not be affected even if the server is made local, so that network communication is unecessary.

4. INTER-AGENT PROTOCOLS

4.1 Starting, Stopping, and Checking Status

The *Coordinator* must be able to configure, start, stop, suspend, resume, and check the status of *ContinuousFeedIndexer*, *ContinuousMotionConveyor*, and *ProcessStation* agents and the machinery they control. Consequently, the latter classes are derived from an abstract base class *Controller* whose interface includes these operations. *Controller* also provides a *MakeReady*() operation by which the *Coordinator* can signal a *Controller* agent to prepare its device for starting, if necessary. Class *Coordinator* provides a *ControllerReady*() callback method by which *Controller* agents may inform the *Coordinator* that they are ready for operation.

4.2 Cell Routing and Tracking

A given PS may have multiple possible logical successors. *Coordinator* and *ProcessStation* agents must therefore cooperate in the routing and tracking of cells. A *ProcessStation* agent must be notified about the presence, identification, and status of cells that are approaching the PS it controls, and the results of processing a cell must be sent to the *Coordinator* to be recorded.[4] An object of class *CellNotice* is passed to a *ProcessStation* agent, via the agent's *HandleCellNotice*() method, to notify it about an oncoming cell. For reasons that are discussed in Section 4.3, this is done even if the cell should not be processed by the agent's PS. A *ProcessStation* agent passes an object of class *WorkReport* to the *Coordinator*, via the latter's *HandleWorkReport*() method, to report the results of processing a cell at a PS.[5] Normally, a PS will not process a cell unless the *ProcessStation* agent controlling it has received a *CellNotice* indicating that it should do so. The exception is that a disposal station must reject any cell for which its *ProcessStation* agent has not received a *CellNotice*, since the status of such cells is unknown to the agent. Note that a missing *CellNotice* is an error condition; when a disposal module should reject a particular cell, the *ProcessStation* agent controlling the station will receive a *CellNotice* indicating this.

Different cells in a the same run may have different itineraries, due to the following:

- A processing step or quality test may fail, in which case the affected cell is not processed further but is removed by a disposal station
- Test cells may be sent to a different output station than ordinary cells
- A PS may fail and be replaced on-the-fly by a backup station

---

[3] For example, stream sockets might be replaced with zero-copy or datagram sockets.
[4] This information could be recorded locally if PS controllers had permanent storage.
[5] Note that a single class could be used to encapsulate all data associated with a cell, and instances of this class could be transmitted between the *Coordinator* and *ProcessStation* agents at each processing step. However, this alternative entails unnecessary network traffic, because much cell data need not be transmitted to all *ProcessStation* agents.

Where and by which agents *CellNotice*s should be issued and *WorkReports* should be processed are major design decisions, which affect the maintainability and scalability of the ZZZ software. To clarify these issues, we distinguish four alternative protocols for the communication of *CellNotices* and *WorkReports*: the *broadcast*, *PS-chain*, *centralized*, and *hybrid* protocols.

In the **broadcast** protocol, each *ProcessStation* agent broadcasts *WorkReport*s to all other *ProcessStation* agents and to the *Coordinator*. A *ProcessStation* agent receiving a *WorkReport* examines it to determine whether to process the cell it refers to. The agent will generally process the cell if the *WorkReport* was issued by the agent controlling the PS that precedes its own and if the cell was processed successfully there. The *Coordinator* initiates processing of a cell by broadcasting a dummy *WorkReport* for it. The *Coordinator* simply records the *WorkReport*s it receives. The main advantage of the broadcast protocol is that activating or deactivating a PS does not entail opening or closing of network connections. This does not compensate for its main disadvantage: it requires *ProcessStation* agents to examine *WorkReport*s that are not relevant to them.

The other three protocols discussed here are examples of **subscribe/notify** protocols, in which one agent subscribes or registers with another to receive timely notifications of pertinent events. The subscribing agent establishes a standing request for notifications, thereby eliminating the overhead of requesting each notification individually.

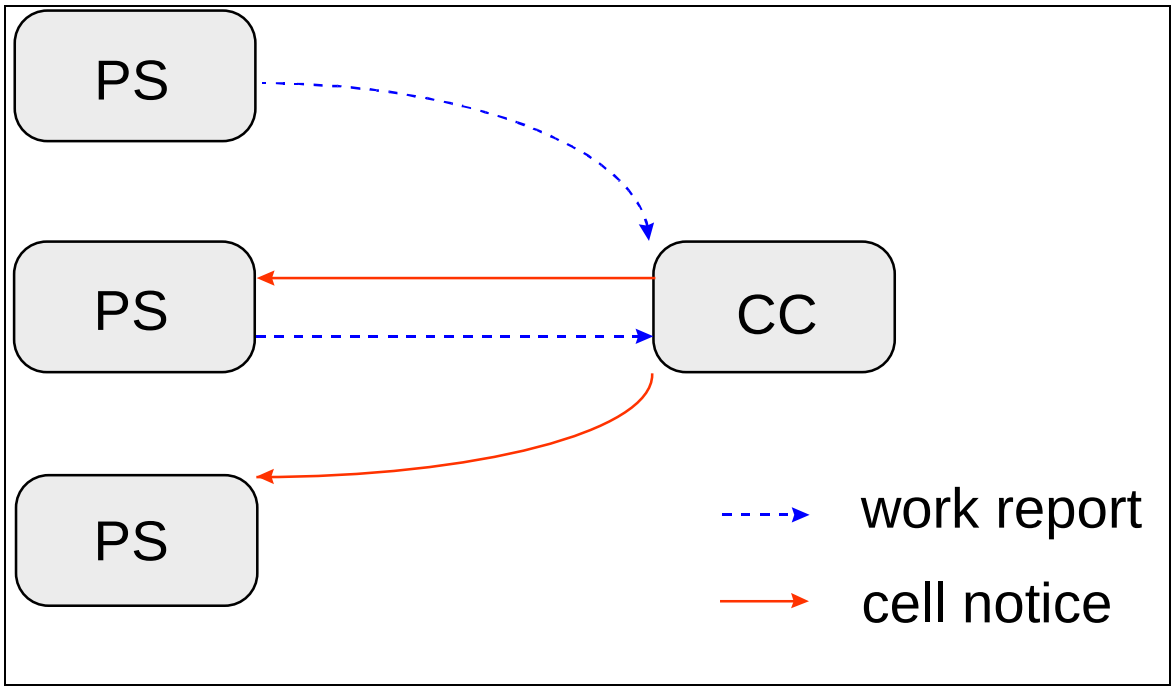

Figure 2: Centralized Protocol

In the **centralized** protocol, each *ProcessStation* agent sends *WorkReport*s to the *Coordinator* (see Figure **2**). The *Coordinator* examines these to determine which PSs should handle the corresponding cells next, and it directs appropriate *CellNotice*s to the *ProcessStation* agents associated with those PSs. The *Coordinator* also records *WorkReport*s. Since the *Coordinator* monitors the status of the entire ZZZ line, the centralized protocol facilitates changing a cell's itinerary in response to line status changes such

as PS failure. A possible drawback of the centralized protocol is overloading of the *Coordinator* agent. With this protocol, the number of *CellNotice*s that the *Coordinator* must issue is proportional to the number of *ProcessStation* agents. This might become excessive if there are a large number of PSs and if the *Coordinator* must do significant computation per *CellNotice*. Neither of these possibilities seems likely at present.

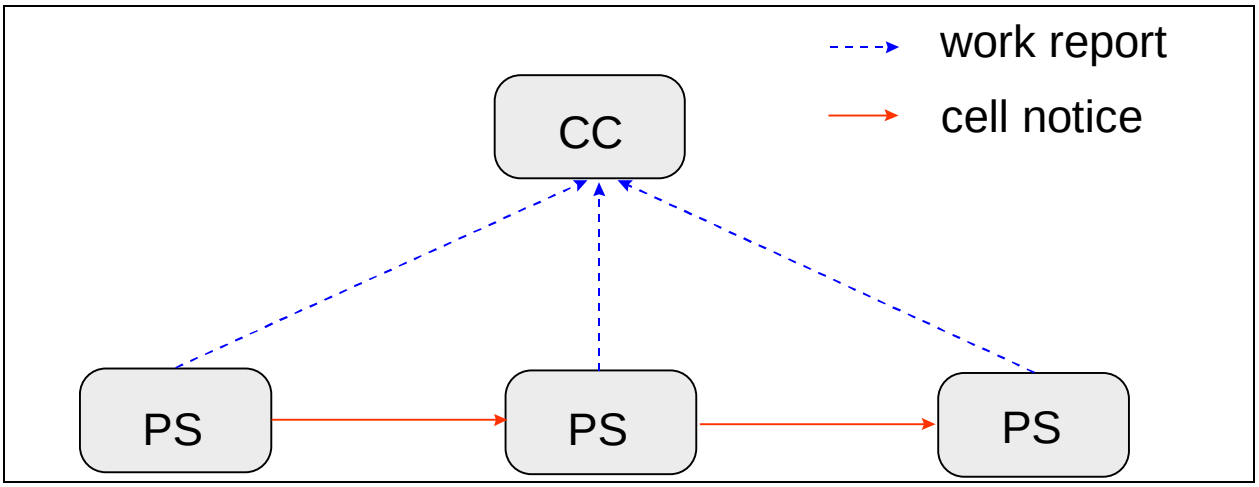

Figure 3: PS-Chain Protocol

In the **PS-chain** protocol each *ProcessStation* agent determines whether a cell it has just processed should be processed further or rejected, and the agent sends a *CellNotice* directly to the appropriate PS or disposal station, without the intervention of the *Coordinator* (see Figure **3**). *ProcessStation* agents do send copies of their *WorkReports* to the *Coordinator* for recording. By distributing responsibility for cell-routing, the PS-chain protocol reduces the *Coordinator*'s workload. However, the *Coordinator* must inform *ProcessStation* agents of line status changes that affect routing. In order for *ProcessStation* agents to route cells as required by the PS-chain protocol, it is necessary for them to know something about the line configuration. This is undesirable, because the implementer of a particular *ProcessStation* agent could incorporate dependencies upon the existing structure that might become invalid in the future.

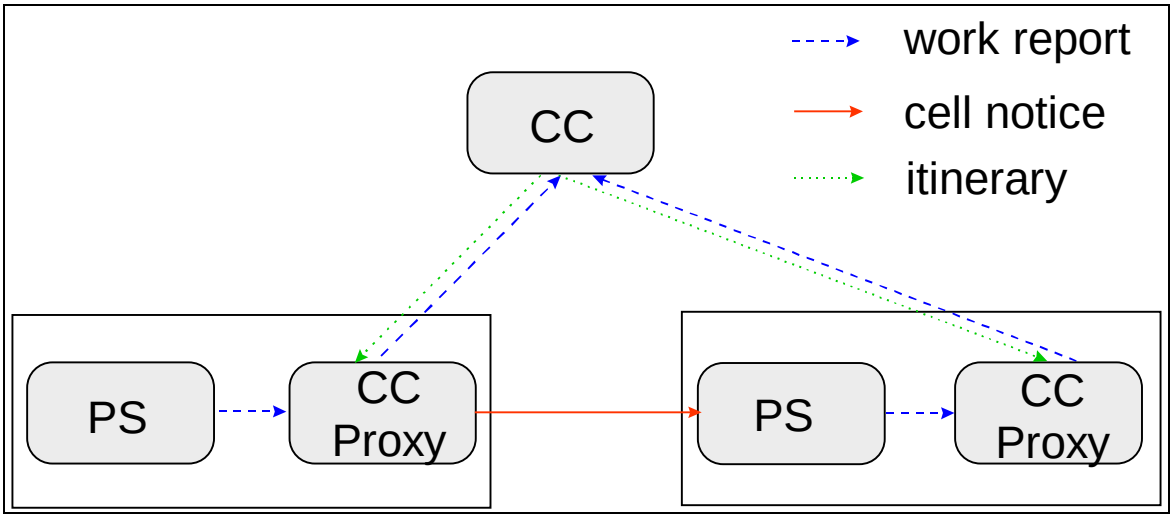

Figure 4: Hybrid Protocol

The protocol we have adopted is a combination of the centralized and PS-chain protocols, which exploits the use of remote proxies. In this **hybrid** protocol, each remote *ProcessStation* agent sends its *WorkReport* for a cell to a local *Coordinator* proxy (see Figure **4**). This proxy determines which PS should process the cell next and issues a *CellNotice* to the *ProcessStation* agent controlling that PS. (The *Coordinator* proxy actually invokes the local proxy for the *ProcessStation* agent, which communicates with the remote agent, although this is not shown in **Figure 1**.) The *Coordinator* proxy does not have to communicate with the real (central) *Coordinator* to route each cell, although the real *Coordinator* must inform the proxy of changes in the itinerary for cells. Thus, the hybrid protocol reduces the workload on the real *Coordinator* but removes knowledge of the line configuration from *ProcessStation* agents.

4.3 Cell Identification

The simplest scheme for tracking cells on the ZZZ line would be to assign each cell a **sequence number** corresponding to its position in the cell stream[6] and to use this number as a cell identifier throughout a cell's processing. A *ProcessStation* agent can identify arriving cells by keeping a count of the cleats that have passed its PS and adding the cleat offset of its PS to this count. This scheme requires that the order of cells on the line be *preserved*. However, there is the possibility that mechanical constraints in the design of a PS will necessitate *reordering* cells.

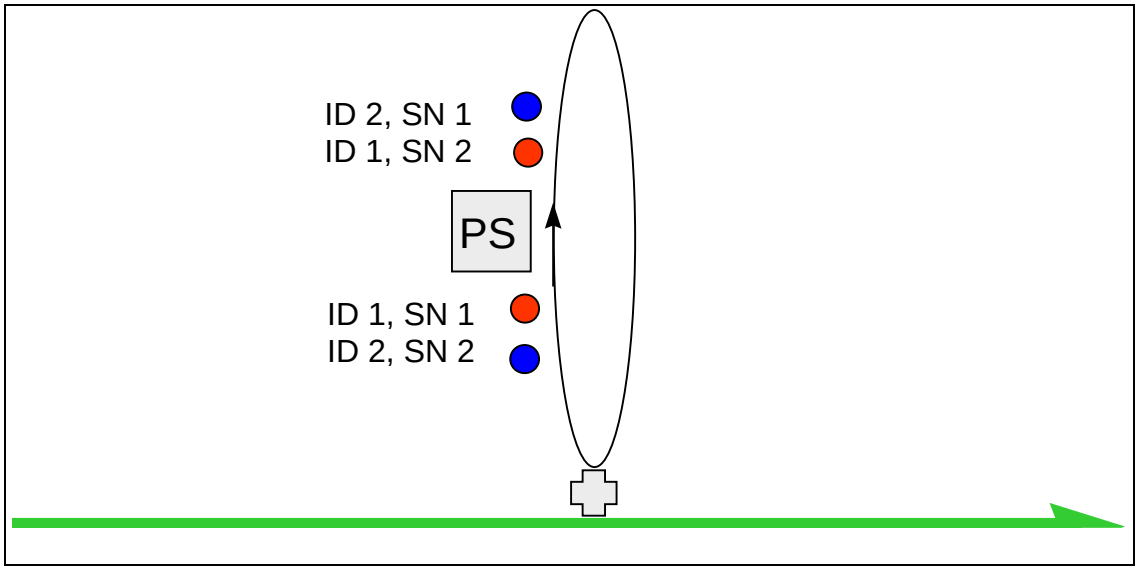

Figure 5: Cell Reordering

As long as this reordering is **deterministic** (not random), it can be accomodated by using *two* numbers to track a cell: a sequence number and a **cell identifier** (**cell ID**). (See Figure **5**.) These numbers are included in *CellNotice* and *WorkReport* objects, and both classes provide *GetSequenceNumber*() and *GetCellID*() methods. The *Coordinator* assigns each cell a cell ID on entry to the system. This ID does not change during the cell's processing, even if cells are reordered. Due to cell reordering by PSs, the sequence number of a cell may change. However, because the reordering is

[6] Note that loss of a cell from its cleat is not considered to alter the sequence numbers of subsequent cells.

deterministic, the *ProcessStation* agent of the PS which does the reordering can inform the *Coordinator* of the change, via the *WorkReport* it issues for the cell. The *Coordinator* can then inform the *ProcessStation* agent controlling the next PS on the line of the cell's new sequence number, via the *CellNotice* it sends to that agent (see Figure **6**).[7] Note that a *ProcessStation* agent must receive a *CellNotice* for each cell, even if the cell should not be processed by the PS that the agent controls. This is done to inform the agent of the cell's identifier, so it can inform the *Coordinator* of changes in the cell's status. Note that a PS can change the sequence number of a cell or damage it even if it is not supposed to process the cell.

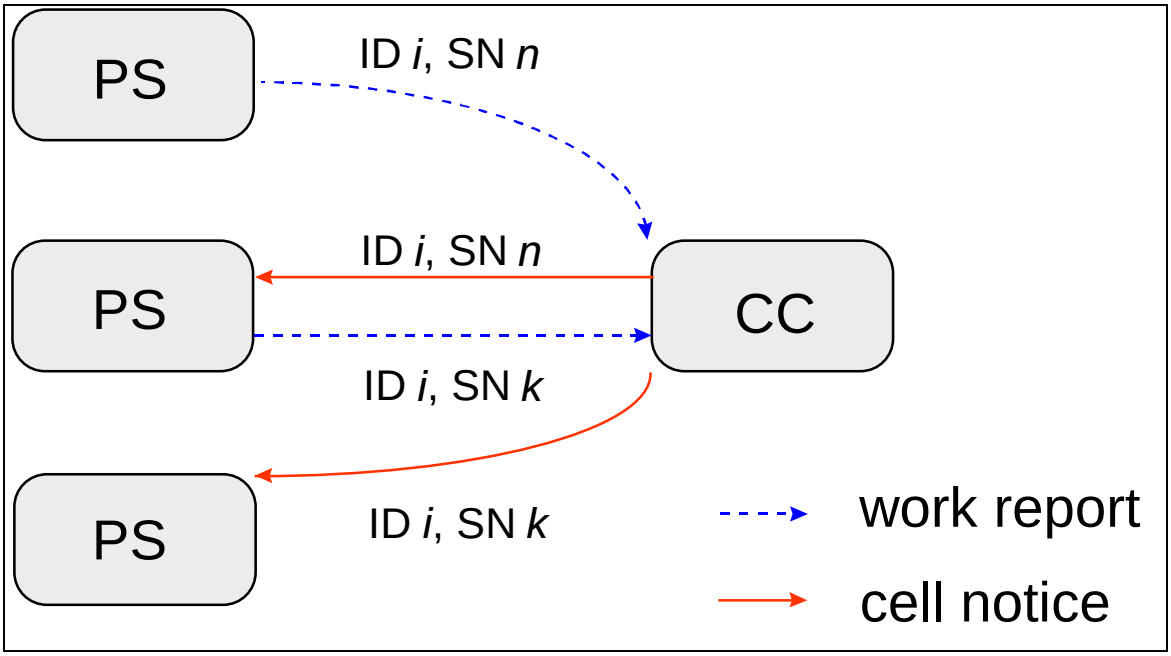

Figure 6: Centralized Protocol with Cell IDs

4.4 Synchronization Between Process Stations and Indexers

The operations of the physical process stations associated with a particular continuous feed indexer must be synchronized with the indexing operations of the CFI. This requires a protocol between *ProcessStation* agents and the *ContinuousFeedIndexer* agent controlling a CFI. A *ContinuousFeedIndexer* provides *RegisterStation*() and *DeregisterStation*() operations which the *Coordinator* uses to tell it which *ProcessStation* agents it must interact with. A *ProcessStation* agent should not actuate its PS until it receives an *Actuate*() message[8] from its *ContinuousFeedIndexer* indicating that the previous indexing operation is complete. Similarly, the *ContinuousFeedIndexer* should not commence an indexing operation until it has received a *StationReady*() message from each *ProcessStation* agent registered with it. Receipt of these messages indicates that it is safe to index. The *ContinuousFeedIndexer* has a time window, determined by the state of its dancer, during which all *ProcessStation* agents must assert *StationReady*(). If a *ProcessStation* agent fails to do so during this window, say because one of its process heads has jammed, the *ContinuousFeedIndexer* agent must report this failure to the *Coordinator*, which in turn will shut down the line.

[7] We assume that the ZZZ line structure implements a "virtual belt" with a single stream of cells. If other line topologies are employed, the inter-agent protocols described here may require modification.
[8] This message was formerly called *doStep*().

4.5 Control of the Continuous Motion Conveyor

In addition to providing the operations of class *Controller*, the *ContinuousMotionConveyor* agent must permit the *Coordinator* to check and, if necessary, alter the speed of the continuous motion conveyor. Hence class *ContinuousMotionConveyor* includes the operations *GetSpeed*() and *ChangeSpeed*().

4.6 Failures and Exceptions

A variety of failures and exceptions (anomalous conditions) may occur in the ZZZ system, for example:

- Device failures (PS, CFI, CMC, etc.)
- Software failures
- Power system failures
- Network failure

The ZZZ control software should recognize and handle such failures to the extent possible. Appropriate responses range from disabling a malfunctioning process head to shutting down the entire system. When an operation in an object-oriented program fails, the operation typically raises or "throws" an exception, which propagates up the chain of active method invocations until a handler is found for the exception. The exception itself is an object, often containing information about the cause of the exception. A hierarchy of exception classes is typically defined to classify the failures and exceptions that are possible in an application. This model of exception handling is appropriate for some ZZZ failures and exceptions but not for others. An exception thrown by a function can be viewed as a special kind of return value. It may be undesirable for a software agent to block waiting to determine if a relatively lengthy operation succeeds or fails. For example, the *HandleCellNotice*() method of a *ProcessStation* agent should not block the *Coordinator* until the corresponding cell is processed.

An alternative to blocking until an operation completes is to require client objects to provide a callback method that the server can invoke asynchronously to inform the client of failures and exceptions. The server may have to pass the client sufficient information to identify which service request failed. This information may of course be embodied in an object. To accomodate both approaches to the handling of failures and exceptions, we define a common failure/exception class hierarchy, rooted at the abstract base class *Failure*, for synchronous and asynchronous failure/exception handling.[9] Agents that must permit servers to notify them asynchronously of failures provide a *HandleFailure*() method for this purpose.[10]

Note that each *Controller* agent must monitor the health of the subsystem it controls. In the event of a failure of that subsystem, it should shut it down and invoke the *HandleFailure*() method of the *Coordinator*.

4.7 Interface to the Safety Circuitry

The safety circuitry of the ZZZ system will remove power to electromechanical devices in the event of an e-stop or a hazard such as an imminent dancer overrun. The control software must respond to a safety shutdown by preserving sufficient information to permit the cause of the shutdown to be determined and

[9] Failures or exceptions specific to a subclass of *Controller* agent should be declared as nested classes within the *Controller* subclass.
[10] This method was formerly called *componentFailure*()

to permit the system to be restarted later with minimal loss of product. Hence, there is at least one *SafetyMonitor* agent that monitors the inputs to the safety circuitry and, in the event of a safety shutdown, records the cause of the shutdown and invokes the *HandleSafetyShutdown*() method of the *Coordinator*. The *HandleSafetyShutdown*() method takes a *Hazard* object describing the cause of the shutdown as an argument; *Hazard* is a subclass of the failure/exception base class *Failure*.

It is possible that the *Coordinator* or a *Controller* agent will need to initiate a safety shutdown, e.g., because the *Coordinator* cannot communicate with *ProcessStation*, *ContinuousFeedIndexer* and *ContinuousMotionConveyor* agents due to network failure. Hence, the *SafetyMonitor* agent has a *Shutdown*() method for this purpose. Although the decision has not been finalized, it may be desirable to have a *SafetyMonitor* agent running on the host of each *Controller* agent, so that each agent can be directly alerted of or directly initiate safety shutdowns.

4.8 Interaction Diagrams

The following interaction diagrams describe communication between the principal software agents in various operational scenarios.

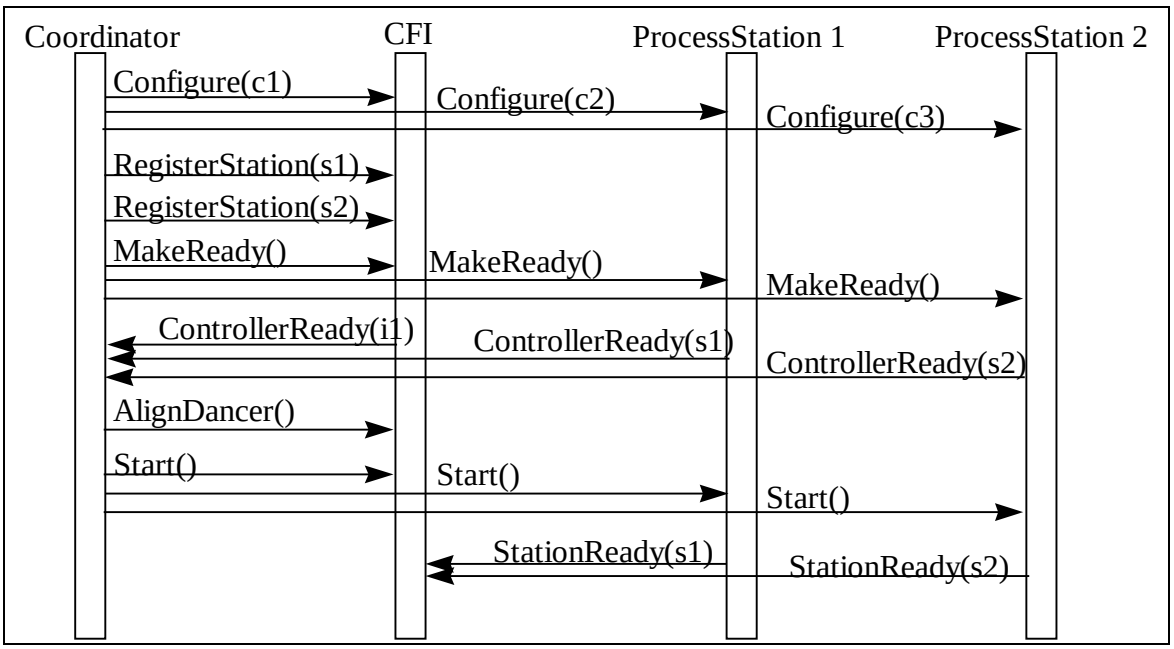

Figure 7: Startup

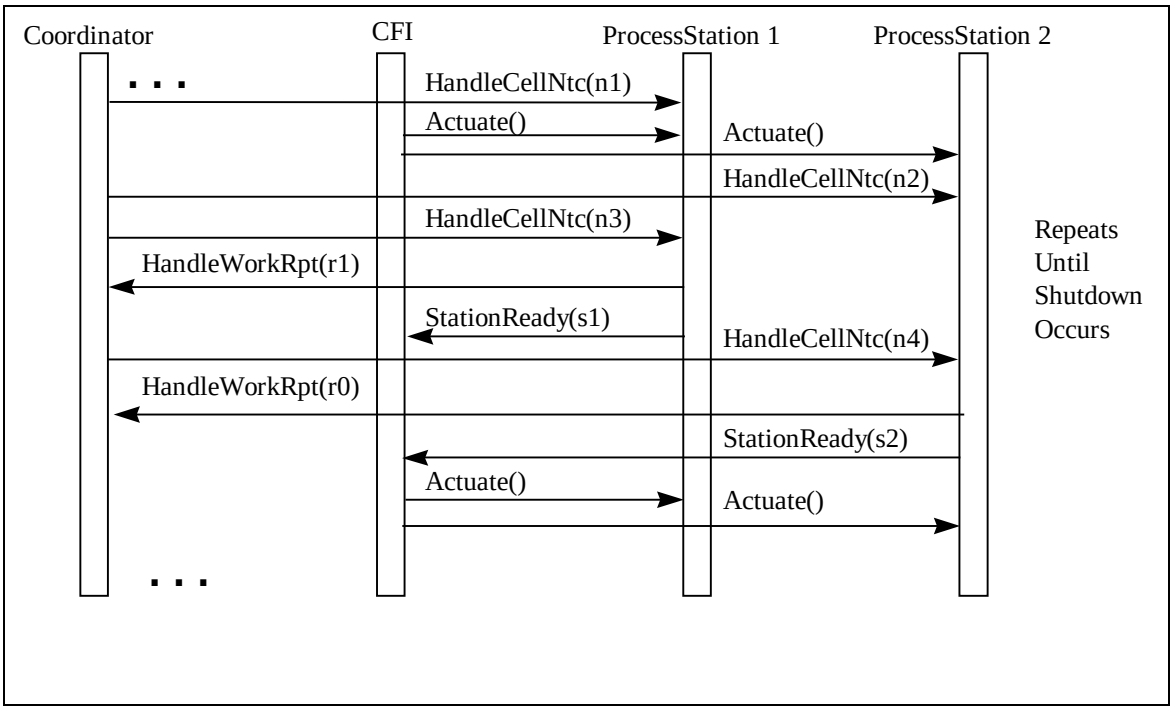

Figure 8: Normal Operation

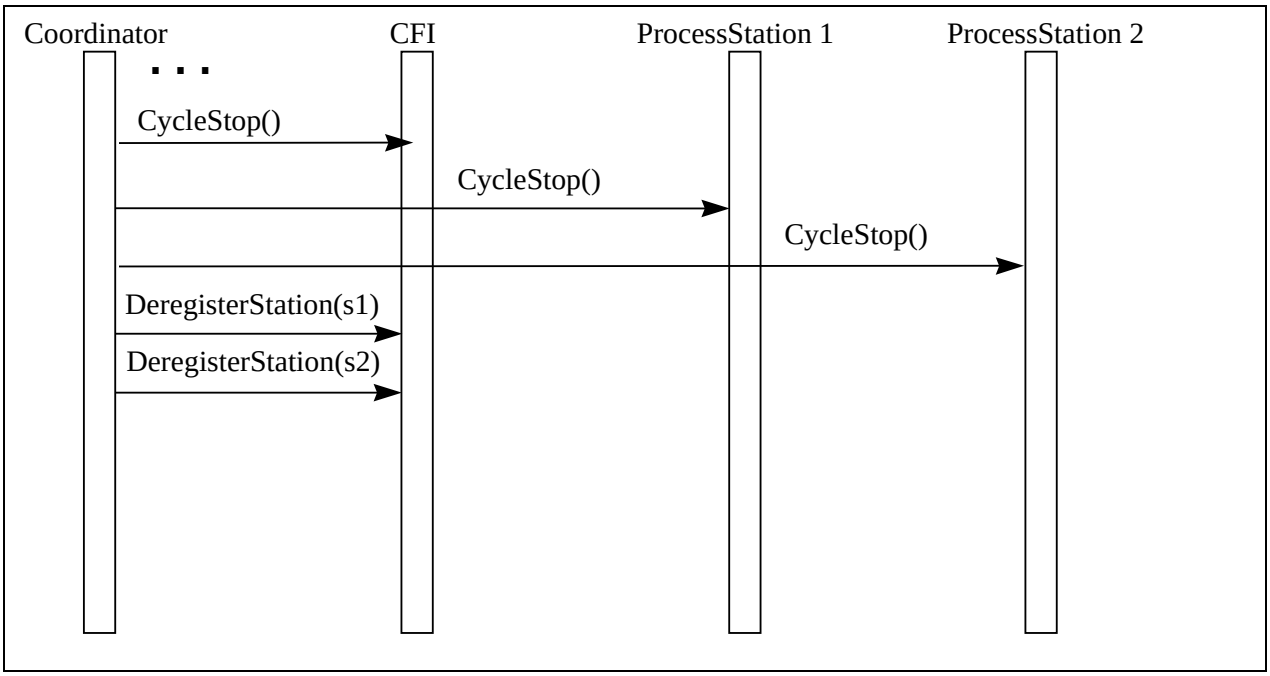

Figure 9: Normal Shutdown of Line (Cycle Stop)

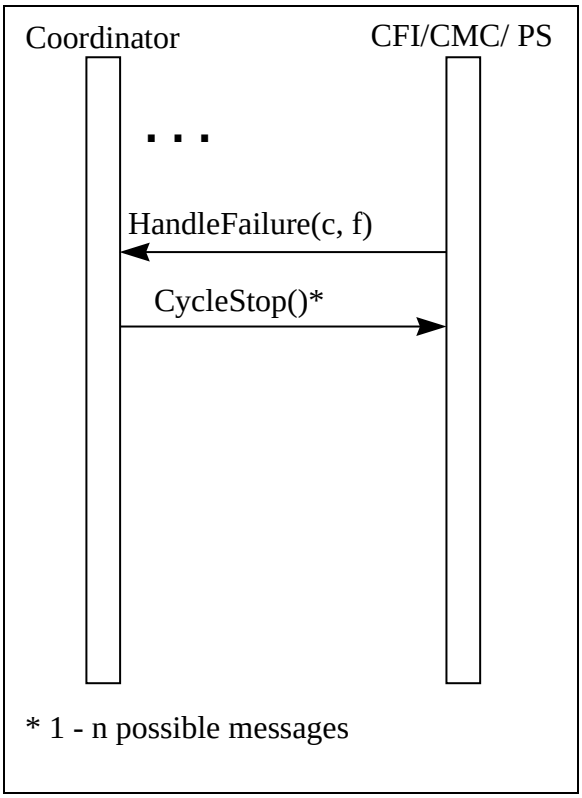

Figure 10: Shutdown Initiated by Controller

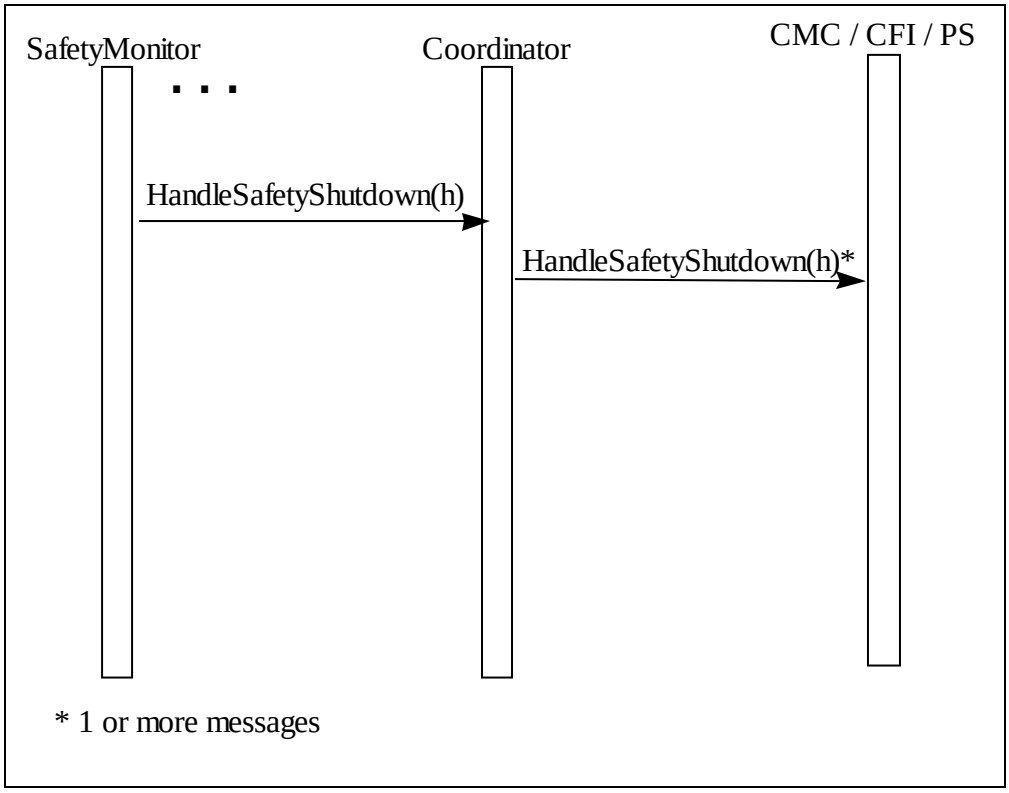

Figure 11: Shutdown Initiated by Safety Circuitry

5. CLASS INTERFACES

This section describes the interfaces of the object classes identified to date in the ZZZ software architecture, including principal and auxiliary classes. These interfaces are incomplete and subject to revision. Constructors, destructors, and exceptions are currently omitted. Classes are listed in alphabetical order.

5.1 Class Cell

An instance of class *Cell* represents the state of a cell that is being processed. *Cell* objects are used internally by the *Coordinator*.

5.1.1 Public Method GetBatch

CellBatch& *GetBatch*()

The *GetBatch*() method returns a cell's batch.

5.1.2 Public Method GetHistory

CellHistory& *GetHistory*()

The *GetHistory*() method returns a cell's processing history.

5.1.3 Public Method GetID

CellIdentifier GetID()

The *GetID*() method returns a cell's unique identifier.

5.1.4 Public Method IsBad

bool *IsBad*()

The *IsBad*() method returns *true* if a fully or partially completed cell fails to meet specifications or if its quality cannot be determined. Bad cells are subject to disposal.

5.1.5 Public Method MarkBad

void *MarkBad*()

The *MarkBad*() method marks a cell as bad.

5.2 Class CellBatch

An instance of class *CellBatch* represents a batch of cells under production. Although the current requirements for the ZZZ do not address batches, they may eventually be required.

5.2.1 Public Method GetCell

Cell GetCell*(*CellIdentifier*)

The method call *b.GetCell*(*cid*) returns the *Cell* object in batch *b* whose *CellIdentifier* is *cid* if such a cell exists; otherwise, it returns *null*.

5.2.2 Public Method GetItinerary

CellItinerary& *GetItinerary*()

The *GetItinerary*() method returns the intended itinerary of all cells in a batch.

5.2.3 Public Method GetRecipe

CellRecipe& *GetRecipe*()

The *GetRecipe*() method returns the recipe used to produce all cells in a batch.

5.3 Class CellHistory

An instance of class *CellHistory* represents a cell's processing history — the sequence of *WorkReport*s about a cell issued by the *ProcessStation* agents that processed it. The methods of class *CellHistory* permit these *WorkReport*s to appended to the sequence and to be accessed in the order that they were appended, which should be the same order in which they were issued. *CellHistory* objects maintain an internal cursor whose position is modified by the *GetFirstReport*() and *GetNextReport*() methods. Initially, the cursor is positioned at the start of a cell's history.

5.3.1 Public Method AddReport

void *AddReport*(*WorkReport*& *r*)

The *AddReport*() method appends a *WorkReport* to the processing history of a cell.

5.3.2 Public Method GetFirstReport

WorkReport GetFirstReport*()

The *GetFirstReport*() method returns the first *WorkReport* in a cell's processing history and positions the cursor at that *WorkReport*; it returns *null* if the history is empty.

5.3.3 Public Method GetNextReport

WorkReport GetNextReport*()

The *GetNextReport*() method returns the next *WorkReport* in a cell's processing history after the one the cursor points to and advances the cursor, provided the history is not empty and the cursor is not initially positioned at its end. If the history is empty or the cursor points to the last *WorkReport* in the history when *GetNextReport*() is invoked, the method returns *null*.

5.4 Class CellIdentifier

An instance of class *CellIdentifier* represents the unique identifier used throughout the system to identify a particular cell. The interface of *CellIdentifier* is not yet defined.

5.5 Class CellItinerary

An instance of class *CellItinerary* represents a cell's itinerary — the sequence of process stations at which a cell is intended to be processed, provided the cell is processed correctly and no PSs fail. *CellItinerary* objects maintain an internal cursor whose position is modified by the *GetFirstStation*() and *GetNextStation*() methods. Initially, the cursor is positioned at the start of a cell's itinerary.

5.5.1 Public Method AddStation

void *AddStation*(*ProcessStation*& *m*)

The *AddStation*() method appends a *ProcessStation* object to the end of a partial itinerary and positions the cursor at that object.

5.5.2 Public Method GetFirstStation

ProcessStation GetFirstStation*()

The *GetFirstStation*() method returns the first *ProcessStation* object in a cell's itinerary and positions the cursor at that object; it returns *null* if the itinerary is empty.

5.5.3 Public Method GetNextStation

ProcessStation GetNextStation*()

The *GetNextStation*() method returns the next *ProcessStation* object in a cell's itinerary after the one the cursor points to and advances the cursor, provided the itinerary is not empty and the cursor is not initially positioned at its end. If the itinerary is empty or the cursor points to the last *ProcessStation* object in the itinerary when *GetNextReport*() is invoked, the method returns *null*.

5.6 Class CellNotice

The *Coordinator* passes an instance of class *CellNotice* as an argument to a *ProcessStation* agent's *HandleCellNotice*() method to notify the agent of whether and how it should process a cell approaching its PS and to inform it of the cell's (global) identifier.

5.6.1 Public Method GetID

CellIdentifier GetID()

The *GetID*() method returns the (global) identifier of a cell.

5.6.2 Public Method GetSequenceNumber

SequenceNumber GetSequenceNumber()

The *GetSequenceNumber*() method returns the sequence number of the cleat in which the cell will arrive at the PS.

5.6.3 Public Method IsBad

bool *IsBad*()

The *IsBad*() method returns *true* if the approaching cell has been marked bad and it returns *false* otherwise.

5.7 Class CellRecipe

An instance of class *CellRecipe* indicates how a batch of cells should be processed. The interface of *CellRecipe* is not yet defined.

5.8 Class ContinuousFeedIndexer (extends Controller)

An instance of class *ContinuousFeedIndexer* is the controlling agent of a CFI. It is responsible for indexing the CFI and for actuating its associated PSs.

Class *ContinuousFeedIndexer* extends the public interface of class *Controller* with the following.

5.8.1 Nested Class IndexerConfiguration (extends Controller::Configuration)

An instance of class *IndexerConfiguration* represents a specification used to configure a *ContinuousFeedIndexer* agent prior to operation of its CFI. The interface of *IndexerConfiguration* is not yet defined.

5.8.2 Public Method DeregisterStation

void *DeregisterStation*(*ProcessStation*& *s*)

The method call *cfi*.*DeregisterStation*(*s*) causes the *ContinuousFeedIndexer* agent *cfi* to remove *ProcessStation* agent *s* from its list of registered *ProcessStation* agents. This is a nonblocking call.

5.8.3 Public Method StationReady

void *StationReady*(*ProcessStation*& *s*)

The method call *cfi*.*StationReady*(*s*) informs *ContinuousFeedIndexer* agent *cfi* that *ProcessStation* agent *s* has completed its previous actuation successfully and is prepared for *cfi* to index. This is a nonblocking call.

5.8.4 Public Method RegisterStation

void *RegisterStation*(*ProcessStation*& *s*)

The method call *cfi*.*RegisterStation*(*s*) causes the *ContinuousFeedIndexer* agent *cfi* to add *ProcessStation* agent *s* to its list of registered *ProcessStation* agents, so that *s* receives *Actuate*() signals from *cfi* and is expected to send *StationReady*() signals to *cfi*. *RegisterStation*() is a nonblocking call.

5.9 Class ContinuousMotionConveyor (extends Controller)

An instance of class *ContinuousMotionConveyor* is the controlling agent of a CMC. It is responsible for starting, stopping, and changing the speed of the CMC on command of the *Coordinator*.

Class *ContinuousMotionConveyor* extends the public interface of class *Controller* with the following classes and methods.

5.9.1 Nested Class ConveyorConfiguration (extends Controller::Configuration)

An instance of class *ConveyorConfiguration* represents a specification used to configure a *ContinuousMotionConveyor* agent prior to operation of its CMC. The interface of *ConveyorConfiguration* is not yet defined.

5.9.2 Nested Class ConveyorSpeed

An instance of class *ConveyorSpeed* represents a valid continuous motion conveyor speed setting. The interface of class *ConveyorSpeed* is not yet defined.

5.9.3 Public Method ChangeSpeed

void *ChangeSpeed*(*ConveyorSpeed* s)

The call *ChangeSpeed*(*s*) causes a *ContinuousMotionController* agent to change the speed of its CMC to *s*. This is a nonblocking call.

5.9.4 Public Method GetSpeed

ConveyorSpeed GetSpeed()

The *GetSpeed*() method returns the current speed of the CMC controlled by a *ContinuousMotionController* agent.

5.10 Class Controller

Class *Controller* is the (abstract) base class for agents controlling ZZZ devices. In particular *ContinuousFeedIndexer*, *ContinuousMotionConveyor*, and *ProcessStation* are derived from *Controller*.

5.10.1 Nested Class Configuration

An instance of class *Configuration* represents a specification used by a controller to configure a device prior to operation. Subclasses of *Configuration* may be defined for different types of devices. The interface of class *Configuration* is not yet defined.

5.10.2 Nested Class Status

An instance of class *Status* represents a description of the status of a controller's device. Subclasses of *Status* may be defined for different types of devices. The interface of class *Status* is not yet defined.

5.10.3 Public Method Configure

void *Configure*(*Configuration*& *c*)

The call *Configure*(*c*) causes a controller agent to configure its device for operation according to the configuration *c*. The *Configure*() call should be made to a *Controller* agent only at system startup or after the *CycleStop*() message has been sent to it. *Configure*() is a nonblocking call.

5.10.4 Public Method CycleStop

void *CycleStop*()

The *CycleStop*() call causes a controller to stop operation of its device. This is a nonblocking call.

5.10.5 Public Method GetStatus

void *Status*& *GetStatus*()

The *GetStatus*() method returns the operational status of a device.

5.10.6 Public Method MakeReady

void *MakeReady*()

The *MakeReady*() method is called by the *Coordinator* to signal a *Controller* agent to prepare to start its device. When ready, the agent will make a *ControllerReady*() call to the *Coordinator*. This handshaking is required when a device cannot start immediately, e.g., because heating is necessary. *MakeReady*() is a nonblocking call.

5.10.7 Public Method HandleSafetyShutdown

void *HandleSafetyShutdown*(*Hazard*& *h*)

The call *HandleSafetyShutdown*(*h*) causes a controller agent to respond to a safety shutdown of the system, caused by the hazard *h*. This is a nonblocking call.

5.10.8 Public Method Resume

void *Resume*()

The *Resume*() method resumes operation of a device whose operation was previously suspended. It is not necessary to call *MakeReady*(), provided *CycleStop*() has not been called. *Resume*() is a nonblocking call.

5.10.9 Public Method Start

void *Start*()

The *Start*() call causes a controller to start operation of its device. This is a nonblocking call.

5.10.10 Public Method Suspend

void *Suspend*()

The *Suspend*() call causes a controller to temporarily suspend operation of its device. Operation can be resumed, without calling *MakeReady*(), by calling *Resume*(). *Suspend*() is a nonblocking call.

5.11 Class Coordinator

Class *Coordinator* has a single instance which controls the ZZZ line.

5.11.1 Public Method ControllerReady

void *ControllerReady*(*Controller*& *c*)

The call *ControllerReady*(*c*) informs the *Coordinator* that *Controller* agent *c* is ready for operation, pending a *Start*() message from the *Coordinator*. *ControllerReady*() is a nonblocking call.

5.11.2 Public Method HandleFailure

void *HandleFailure*(*Failure*& *f*)

A *Controller* agent sends the message *HandleFailure*(*f*) to the *Coordinator* to inform it that the failure *f* has occurred in the *Controller*'s subsystem. The argument *f* should indicate the source of the message. *HandleFailure*() is a nonblocking call.

5.11.3 Public Method HandleSafetyShutdown

void *HandleSafetyShutdown*(*Hazard*& *h*)

The *SafetyMonitor* sends the message *HandleSafetyShutdown*(*h*) to the *Coordinator* to inform it of a safety shutdown of the system, caused by hazard *h*. *HandleSafetyShutdown*() is a nonblocking call.

5.11.4 Public Method HandleWorkReport

void *HandleWorkReport*(*WorkReport*& *r*)

A *ProcessStation* agent sends the message *HandleWorkReport*(*r*) to the *Coordinator* to report the results of processing a particular cell.[11] The argument *r* should indicate the source of the message. *HandleWorkReport*() is a nonblocking call.

5.12 Class Dancer

Class *Dancer* models the dancer which couples a CFI to the CMC.

5.12.1 Enumeration Type Direction

enum *Direction* {*LEFT, RIGHT*}

The enumeraton type *Direction* represents the two possible directions of dancer movement, as observed from its CFI.

[11] It may be useful for debugging purposes for a *ProcessStation* agent to issue *WorkReport*s even for cells that it does not process because they were previously marked as bad.

5.12.2 Nested Class Position

An instance of class *Position* represents the position of the dancer as indicated by the sensors positioned around it. The interface of *Direction* is not yet defined.

5.12.3 Public Method GetDirection

Direction GetDirection()

The *GetDirection*() method returns the latest sensed direction of dancer movement.

5.12.4 Public Method GetPosition

Position GetPosition()

The *GetPosition*() method returns the latest sensed position of the dancer.

5.13 Class Failure

Class *Failure* is the abstract base class for all failures and exceptions. Its interface is not yet defined.

5.14 Class ProcessStation (extends Controller)

An instance of class *ProcessStation* is the controlling agent of a PS. It actuates a set of process heads implementing a step in the cell manufacturing process.

Class *ProcessStation* extends the interface of class *Controller* with the following.

5.14.1 Nested Class StationConfiguration (extends Controller::Configuration)

An instance of class *StationConfiguration* represents the specification used to configure a *ProcessStation* agent prior to operation of its PS. The interface of *StationConfiguration* is not yet defined.

5.14.2 Public Method Actuate

void *Actuate*()

The *Actuate*() method is called by a *ContinuousFeedIndexer* agent to signal a *ProcessStation* agent that it should actuate the process heads of its PS. *Actuate*() is a nonblocking call.

5.14.3 Public Method HandleCellNotice

void *HandleCellNotice*(*CellNotice*& *n*)

The message *HandleCellNotice*(*n*) is sent by the *Coordinator* to a *ProcessStation* agent to indicate that it should process a cell approaching its PS and possibly to indicate how it should be processed. *HandleCellNotice*() is a nonblocking call.

5.15 Class SafetyMonitor

There is likely to be a single instance of class *SafetyMonitor*, which implements the control software's interface to the safety circuitry. This *SafetyMonitor* agent monitors the safety circuitry for a shutdown indication. If such an indication is received the *SafetyMonitor* agent informs the *Coordinator* of the shutdown and its cause by calling the *Coordinator*'s *HandleSafetyShutdown*() method. The *SafetyMonitor* can also receive shutdown requests via its *Shutdown*() method.

5.15.1 Nested Class Hazard (extends Failure)

An instance of class *Hazard* provides information about the cause of a safety shutdown. The interface of class *Hazard* is not yet defined.

5.15.2 Public Method Shutdown

void *Shutdown*()

The *Shutdown*() method causes the *SafetyMonitor* agent to initiate a safety shutdown. *Shutdown*() is a nonblocking call.

5.16 Class SequenceNumber

An instance of class *SequenceNumber* represents the current position of a cell in the cell-stream. (Note that empty cleats are considered to occupy positions in the cell stream.) The interface of this class is not yet defined.

5.17 Class WorkReport

An instance of class *WorkReport* represents the results of processing of a cell by a PS. For each cell it processes, a *ProcessStation* agent sends a *WorkReport* to the *Coordinator* via the latter's *HandleWorkReport*() method.

5.17.1 Public Method GetCellID

CellIdentifier GetCellID()

The *GetCellID*() method returns the unique, global identifier of the cell.

5.17.2 Public Method GetProcessStation

ProcessStation& *GetProcessStation*()

The *GetProcessStation*() method returns the *ProcessStation* agent that processed the cell.

5.17.3 Public Method GetCellNotice

CellNotice& *GetCellNotice*()

The *GetCellNotice*() method returns the *CellNotice* that specified processing of the cell.

5.17.4 Public Method GetOutboundSequenceNumber

SequenceNumber GetOutboundSequenceNumber()

The method *GetOutboundSequenceNumber*() returns the sequence number of the cell as it left the PS, that is, after possible cell-reordering by the PS.

5.17.5 Public Method IsBad

bool *IsBad*()

The *IsBad*() method returns *true* if the cell was not processed successfully, otherwise it returns *false*.

6. CLASS DIAGRAMS

6.1 Controller Class Hierarchy

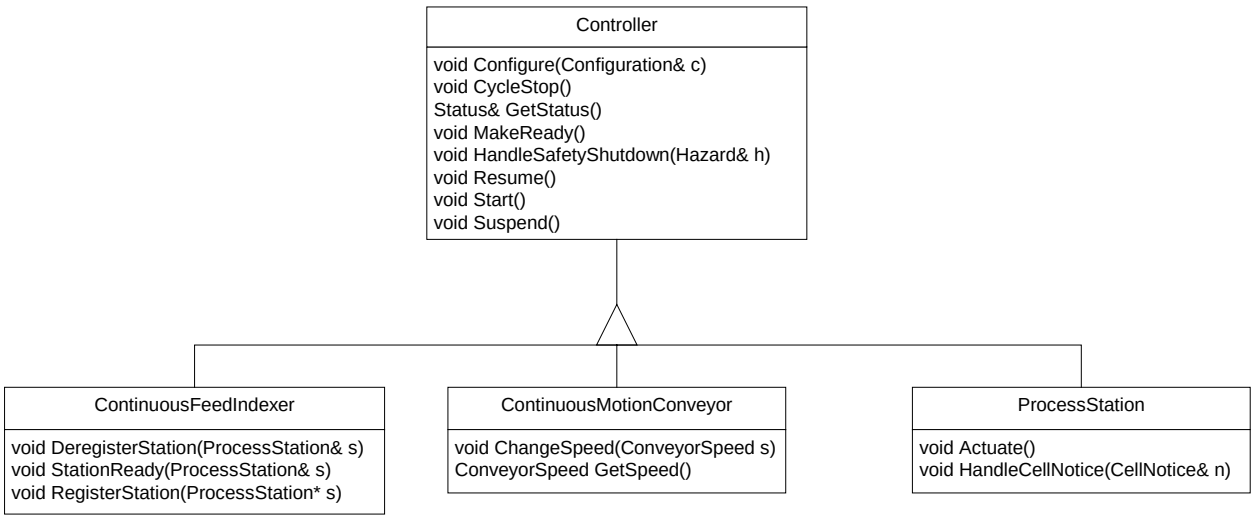

Figure 12: Controller Class Hierarchy

7. TRACEABILITY AND VERIFICATION

Traceability And Verification Table			
Section This Document	Source Section(s)	Validation Technique	Verification Plan