Submitted to PLoP'97

Driver Duty Constructor

Liping Zhao
liping@cs.rmit.edu.au
Department of Computer Science, RMIT, Australia

Ted Foster
ted@class-sc.demon.co.uk
Class Software Construction Ltd., U.K.

ABSTRACT

In this paper, we present a **Driver Duty Constructor** pattern, which is part of our pattern language of transport systems, PLOTS. This pattern consists in the main of three other transport object patterns (TOPS**): Driver Duty Builder** and **Driver Duty Director** which work together to build all the basic parts of a **Driver Duty** and then assemble them into a whole product; the Driver Duty pattern supports a tree structure for explicitly ordering and layering the components that define a driver duty. Each component in a driver duty has one corresponding builder. These Driver Duty Builder objects are represented in perfect symmetry with the Driver Duty objects that they are responsible for building. Both of them use a Cascade of Composite patterns for this purpose. The Driver Duty Director pattern is structured as a Strategy pattern and directs the building process. The interaction of all these contributory patterns defines the overall Driver Duty Constructor pattern, which in turn forms a part of a larger Driver Duty Scheduler system. The emphasis in this paper is on pattern interactions rather than individual patterns in isolation.

INTRODUCTION

"Hsiang Sheng : Nothing functions in isolation; everything functions in relationship with everything else" [Grigg97, preface xxx].

The Driver Duty Constructor pattern consists in the main of three other transport object patterns (TOPS): **Driver Duty Builder** and **Driver Duty Director** which work together to build all the basic parts of a **Driver Duty** and then assemble them into a whole product. The Driver Duty pattern [Zhao+98] supports a tree structure for explicitly ordering and layering the components that define a driver duty. Each component in a driver duty has one corresponding builder. These Driver Duty Builder objects are represented in perfect symmetry with the Driver Duty objects that they are responsible for building. Both use a Cascade of Composite patterns for this purpose. The Driver Duty Director pattern is structured as a Strategy pattern and directs the building process. The interaction of all these contributory patterns defines the overall Driver Duty Constructor pattern, which in turn forms a part of a larger Driver Duty Scheduler system (Figure 1)..

We present our Driver Duty Constructor as a whole using a hybrid pattern form based mainly on the Gang-of-Four and Coplien forms. In other words, within each section we discuss several more basic TOPS together. Sometimes a problem can only be solved by the interaction of two or more closely related patterns. This pattern interaction contributes purpose and behaviour over and above the sum of the individual patterns of which it is composed. We feel that a complex pattern can not necessarily be fully described by discussing each of its more basic patterns in isolation. We want to balance the indubitable benefits of reducing a large complex system into smaller simpler systems against the dangers of losing sight of the whole.

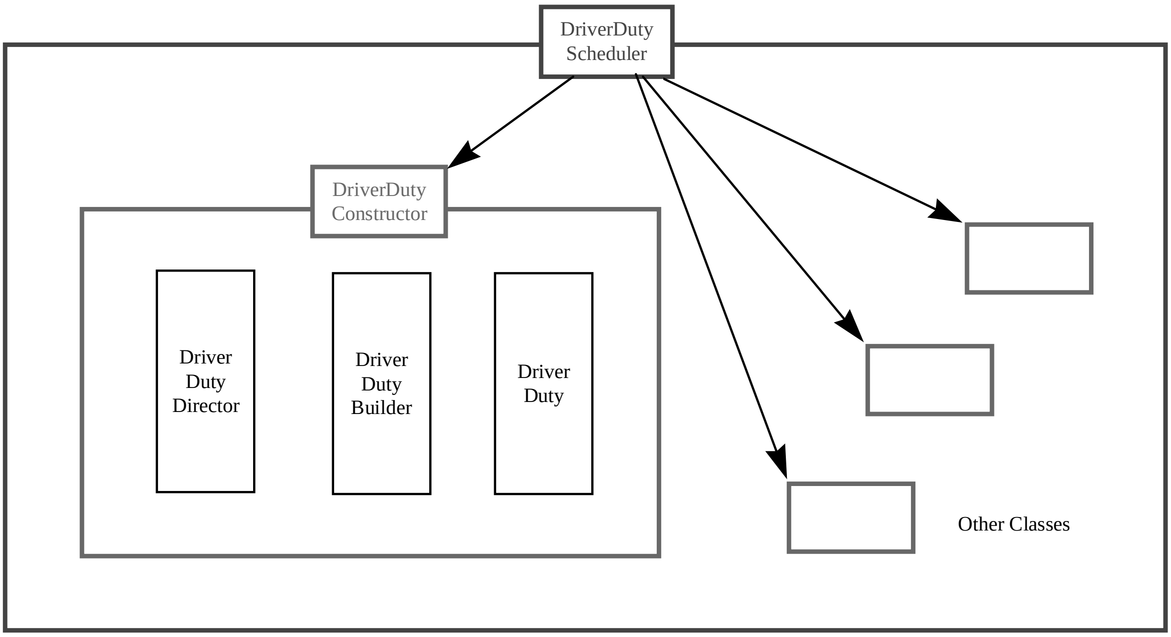

Figure 1. Transport Object Patterns for Driver Duty Scheduling.

DRIVER DUTY CONSTRUCTOR

Intent

The Driver Duty Constructor pattern is for building the basic parts of a driver duty and then assembling them into a whole product to support a variety of scheduling algorithms. It provides clients with a consistent interface to all driver duty components and their compositions. The pattern separates the representation of a driver duty from its construction so that the same construction process can create different duty representations.

Also Known As

Driver Duty also known as Crew Duty or Duty (UK and Europe); Driver Shift, Run or Turn (USA); Workday (Canada); Duty Scheduling (UK and Europe); Run Cutting (USA).

Problem

How do we construct a driver duty?

Let us consider a simple driver scheduling problem in Figure 2. Three driver duties covering work across three vehicle blocks have been formed. A vehicle block contains the work of a vehicle from the time it leaves a parking point until its next return to a parking point. Blocks are partitioned into units of work, which are bounded by driver relief opportunities where drivers can be changed. Driver duty 1 covers the work from 4.50am to 8.45am on vehicle block 3, 9.25am to 12 noon on block 2, and 12.22pm to 13.45pm on block 1; this duty contains a meal break from 8.45am to 9.25am and a join up (a short interval for a driver to transfer between blocks) from 12 noon to 12.22pm. Driver duty 2 covers the work from 4.35am to 9.25am on block 2, and 9.55am to 12.22pm on block 1, with a meal break from 9.25am to 9.55am.

Driver duty 3 covers the work from 5am to 9.55am and then 13.45pm to 1552pm on block 1, with a long break (a period of free time for a driver) from 9.55am to 13.45pm.

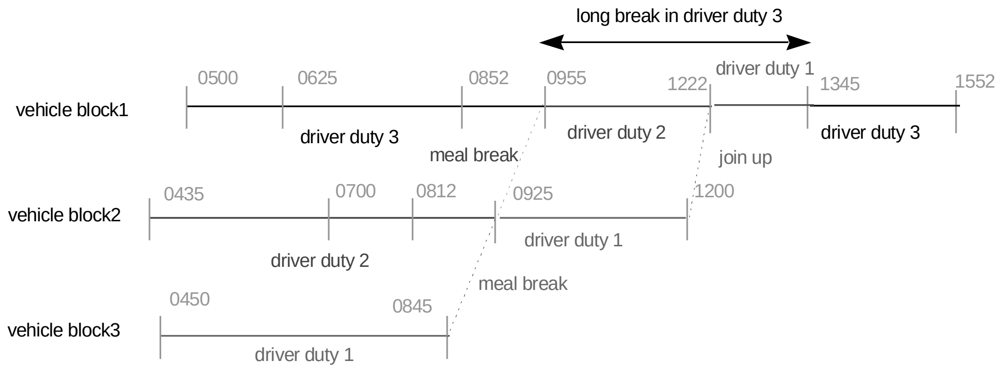

Figure 2. Three driver duties covering work across three vehicle blocks.

Driver duties are made up of driver duty components at a number of abstraction levels. At the bottom level, there are driver spells and driver join ups. A driver spell covers one or more consecutive units of the work on the same vehicle block. For example, in Figure 2, duty 1 has two spells in its second stretch, one from 9.25am to 12 noon on block 2, and the other from 12.22pm to 13.45pm on block 1, with a join up between 12 noon to 12.22pm.

An ordered sequence of these bottom level components makes up a driver stretch at the next level up. A driver stretch is a continuous period of work on one or more vehicle blocks without a driver break (e.g., a meal break). For example, duty 1 in Figure 2 has two driver stretches, one from 4.50am to 8.45am on block 3 and the other from 9.25am on block 2 to 13.45pm on block 1. The second stretch of duty 1 contains two driver spells, and one join up. Other typical components at this level are driver break (e.g., meal break), driver sign-on, driver travel time and driver sign-off.

An ordered sequence of driver stretches and these other components make up driver parts at the next level up. A driver part is a continuous period during which when a driver is in company time. A duty part includes driver breaks, but not driver long breaks. Duty 3 in Figure 2 has two driver parts, one from 5am to 9.55am and the other from 13.45pm to 15.52pm.

An ordered sequence of driver parts (usually two) and driver long breaks (usually only one) make up a complete driver duty. Straight duties have only one driver part; split duties have at least two parts. For example, driver duties 1 and 2 are straight duties with only one driver part made up of two driver stretches and a meal break; whereas duty 3 is a split duty with two driver parts. Figure 3 illustrates the relationship between driver duty 1 and its components.

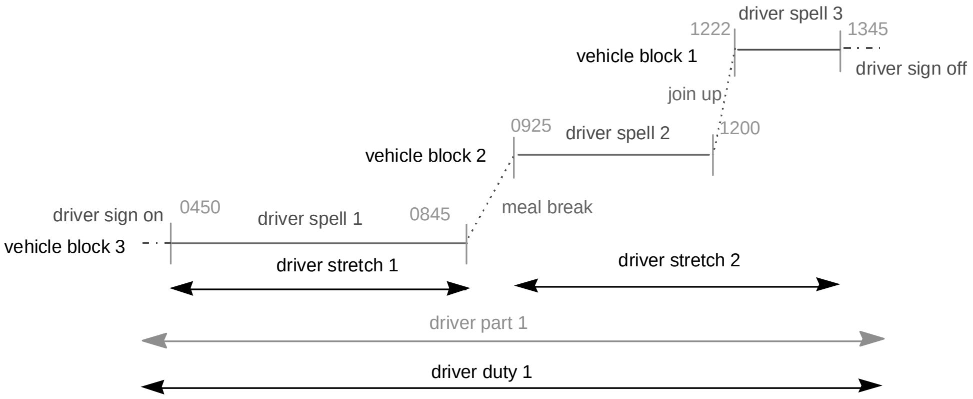

Figure 3. Driver duty 1 and its components.

Because of many differences in labour rules between transport companies, driver duties have many variations. In some companies, a driver duty may have two meal breaks, whereas in others, a morning and afternoon tea break is required. How can we cope with many variations in duties and construct them uniformly?

Solution

– Structure

- Represent driver duty components as a cascade of composites (Figure 4). A driver duty is composed of driver duty components (driver part components and driver long breaks), driver part components (driver stretch components, driver sign on times, driver travel times, driver breaks and driver sign off times), and driver stretch components (driver spells and driver join up times). At each level, an abstract component class provides an interface to a leaf or composite class. The composite classes are ordered aggregates of their parent component class. The three composites are Driver Duty, Driver Part and Driver Stretch. In the special case where a composite class is instantiated with no children, the composite class is in effect behaving as a leaf class.

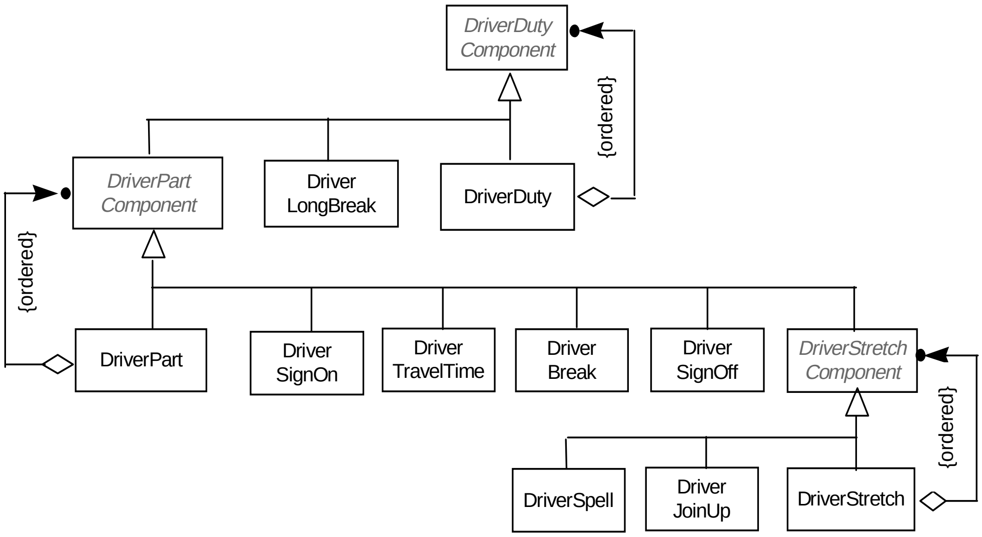

Figure 4. Driver Duty represented as a Cascade of Composites.

- Structure Driver Duty Builder symmetrically with Driver Duty (Figure 5); make each builder responsible for building its corresponding duty component in a particular layer within the duty cascade. (In Figure 5, we only show the major components for both Driver Duty and Driver Duty Builder for simplicity).

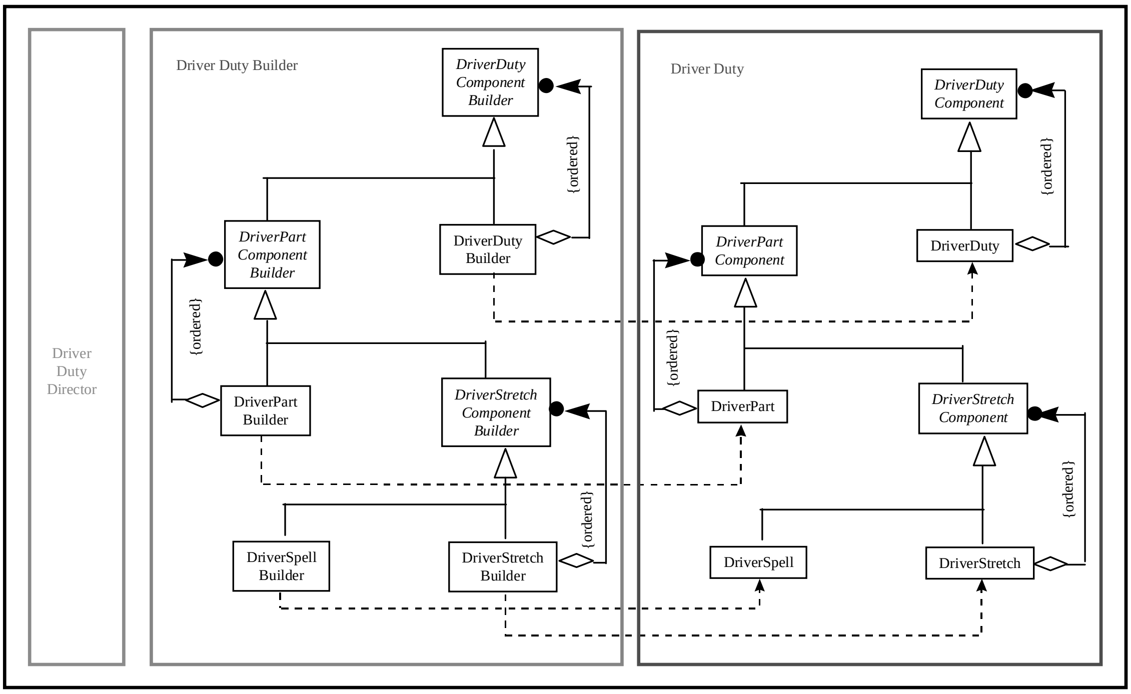

Figure 5. Driver Duty Builder has the same structure as its Products

- Structure Driver Duty Director as a Strategy pattern (Figure 6); each director is responsible for assembling duty components for a particular layer.

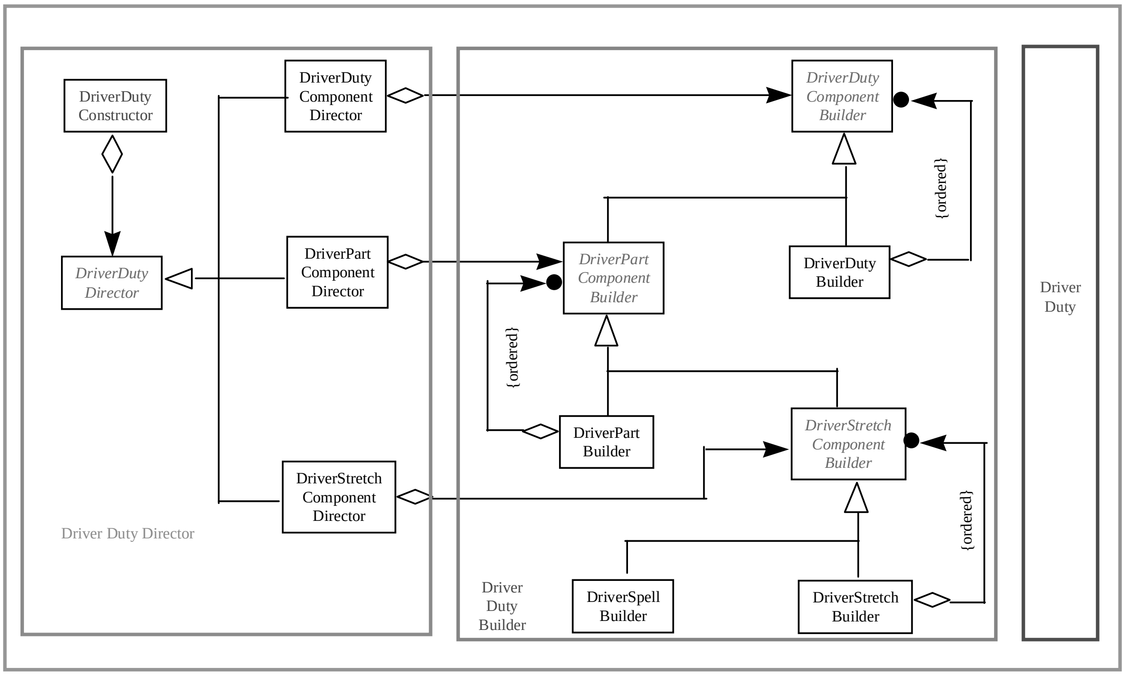

Figure 6. Driver Duty Director plays a more Strategic role than Builder does.

– Example

To illustrate the Driver Duty pattern, let us look at driver duty 1 in Figure 3 and its three main components, i.e., Driver Part, Driver Stretch, and Driver Spell.

1) A driver duty, which is an ordered sequence of one or more driver parts (e.g., driver duty 1 has only one duty part) or driver part components.
2) A driver part, which is an ordered sequence of one or more driver stretches (e.g., driver duty part 1 contains driver stretches 1 and 2) or driver stretch components.
3) A driver stretch, which is an ordered sequence of one or more driver spells (e.g., driver stretch 2 contains driver spells 2 and 3).

Figure 7 shows the object structure of driver duty 1.

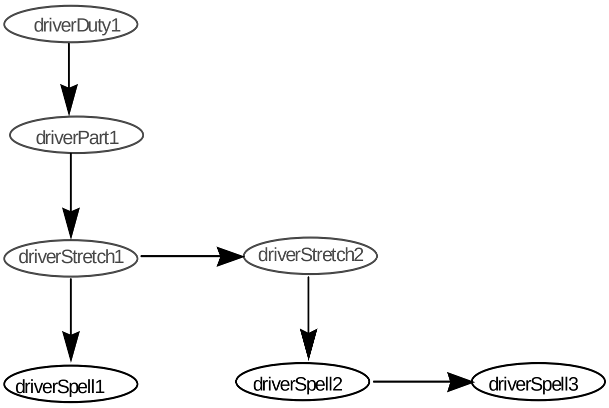

Figure 7. The object structure of a driver duty

The other minor components would appear in their appropriate position in the sequence (e.g., a meal break between the two stretches). Some designers treat these minor components as attributes of the main components, but this makes the main components less reusable by complicating their interfaces. Figure 8 shows three further examples of duties, including more of the minor components (not travel times between different locations), which are all easily catered for by this generic representation.

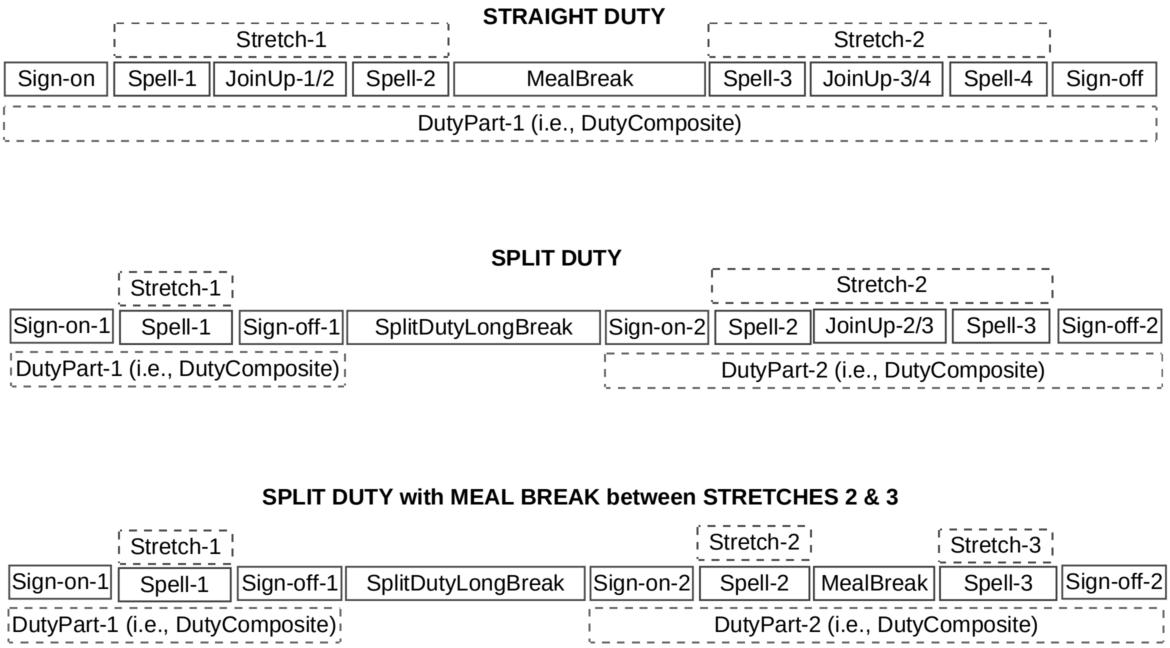

Figure 8. Example of duties and their components

– Participants and Collaborations

Consider one layer at a time from the bottom up.

In the **bottom layer**, spells (and other leaf objects not shown here) are assembled into a driver stretch for use in the middle layer.

- **Driver Stretch Component**
  - Declares a uniform interface for Driver Stretch and Driver Spell objects.
  - Implements default behaviour for the interface common to these Driver Stretch Component objects as appropriate.
  - Declares an interface for managing the ordered sequence of these Driver Stretch Component objects.

- **Driver Spell**
  - Defines behaviour for this leaf object.

- **Driver Stretch (a composite or leaf if not further subdivided)**
  - Defines behaviour for Driver Stretches with zero or more Driver Stretch Component objects.
  - Stores an ordered sequence of Driver Stretch Component objects.
  - Implements operations for managing this ordered sequence of Driver Stretch Component objects.

- **Driver Spell Builder**
  - Builds a driver spell under the direction of Driver Stretch Component Director.
  - Holds general rules that define a valid driver spell.

- **Driver Stretch Builder**
  - Builds a driver stretch under the direction of Driver Stretch Component Director.
  - Makes a sequence of calls to Driver Spell Builder to build the driver spell(s) that make up a driver stretch.
  - Keeps the order in which in which the above calls to Driver Spell Builder are to be made.
  - Holds general rules that define a valid driver stretch.

- **Driver Stretch Component Builder**
  - Defines an abstract interface for Driver Stretch Component Director to build a driver stretch and its driver spell(s).
  - Allows Driver Stretch Component Director to access and manage the sequencing of calls to be made to Driver Spell Builder by Driver Stretch Builder. Declared in the root of the driver stretch component hierarchy for transparency, not safety.

- **Driver Stretch Component Director**
  - Assembles a driver stretch for its client (Driver Part Component Director or Driver Duty Director) using the Driver Stretch Component Builder interface.
  - Hides from its client how driver stretch components are created and assembled into a driver stretch.

In the **middle layer**, driver stretches (and other leaf objects not shown here) are assembled into a driver part for use in the top layer.

- **Driver Part Component**
  - Declares a uniform interface for Driver Part and Driver Stretch Component objects.
  - Implements default behaviour for the interface common to these Driver Part Component objects as appropriate.
  - Declares an interface for managing the ordered sequence of these Driver Part Component objects.

- **Driver Part (a composite or leaf if not further subdivided)**
  - Defines behaviour for Driver Parts with zero or more Driver Part Component objects.
  - Stores an ordered sequence of Driver Part Component objects.
  - Implements operations for managing this ordered sequence of Driver Part Component objects

- **Driver Part Builder**
  - Builds a driver part under the direction of Driver Part Component Director.
  - Makes a sequence of calls to Driver Stretch Component Builder to build the driver stretches that make up a driver part.
  - Keeps the order in which in which the above calls to Driver Stretch Component Builder are to be made.
  - Holds general rules that define a valid driver part.

- **Driver Part Component Builder**
  - Defines an abstract interface for Driver Part Component Director to build a driver part and its driver stretch components.

  - Allows Driver Part Component Director to access and manage the sequencing of calls to be made to Driver Stretch Component Builder by Driver Part Builder. Declared in the root of the driver part component hierarchy for transparency, not safety.

- **Driver Part Component Director**
  - Assembles a driver part for its client (Driver Duty Component Director) using the Driver Part Component Builder interface.
  - Hides from its client how driver part components are created and assembled into a driver part.

At the **top level**, driver parts (and other leaf objects not shown here) are assembled into a driver duty – the final product.

- **Driver Duty Component**
  - Declares a uniform interface for Driver Duty, Driver Long Break, and Driver Part Component objects.
  - Implements default behaviour for the interface common to these Driver Duty Component objects as appropriate.
  - Declares an interface for managing the ordered sequence of these Driver Duty Component objects.

- **Driver Duty (a composite or leaf if not further subdivided)**
  - Defines behaviour for Driver Duties with zero or more Driver Duty Component objects.
  - Stores an ordered sequence of Driver Duty Component objects.
  - Implements operations for managing this ordered sequence of Driver Duty Component objects.

- **Driver Duty Builder**
  - Builds a driver duty under the direction of Driver Duty Component Director.
  - Makes a sequence of calls to Driver Part Component Builder to build the driver parts that make up a driver duty.
  - Keeps the order in which in which the above calls to Driver Part Component Builder are to be made.
  - Holds general rules that define a valid driver duty.

- **Driver Duty Component Builder**
  - Defines an abstract interface for Driver Duty Component Director to build a driver duty and its driver part components.
  - Allows Driver Duty Component Director to access and manage the sequencing of calls to be made to Driver Part Component Builder by Driver Duty Builder. Declared in the root of the driver duty component hierarchy for transparency, not safety.

- **Driver Duty Component Director**
  - Assembles a driver duty for its client (Driver Duty Scheduler) using the Driver Duty Component Builder interface.
  - Hides from its client how driver duty components are created and assembled into a driver duty.

The following classes are **not within any one layer**:

- **Driver Duty Director**
  - Defines an abstract interface common to all directors. Driver Duty Constructor uses this interface to call the algorithm defined by a concrete director.

- **Driver Duty Constructor**
  - Knows which concrete directors are responsible for constructing each layer that is requested by a client. Constructing an upper layer (e.g., driver duty) implies also the construction of its lower layers (duty part and duty stretch) after building all their basic components. Lower layers can be constructed independently of the higher layers.
  - Calls a director via the interface of Driver Duty Director.
  - May define an interface that lets Driver Duty Director access its data.
  - Handles work assigned by a Driver Duty Scheduler, but has no references to this object.

- **Driver Duty Scheduler**
  - Knows about Driver Duty Constructor and other system classes (not described in this paper) which make up the Driver Duty Scheduler system. Driver Duty Scheduler is a Facade object and directs client requests to the main system classes.

Context

Use the Driver Duty Constructor pattern to get the benefit of its three constituent TOPS and more control over the construction process by making its structure symmetric with the product that is under construction.

Use the Driver Duty pattern to order and layer the components that define a driver duty. Clients of a duty can ignore the difference between composite and primitive objects and treat all objects used for duty definition uniformly.

Use the Driver Duty Builder and Director patterns when the algorithm for creating a driver duty should be independent of its components and how they are assembled; the building process must allow different representations for the duty that is constructed.

Pattern Interactions

- **Cascade**
  The underlying pattern of Driver Duty and Driver Duty Builder is a Cascade of Composites; each layer is represented by one Composite [Gamma+95, p. 163].

- **Chain of Responsibility**
  Driver Duty Component Builder, Driver Part Component Builder, and Driver Stretch Component Builder form a Chain of Responsibility [Gamma+95, p. 223]. For example, when Driver Duty Component Builder receives a request from Driver Duty Component Director for implementing a driver duty, it will pass on the request to Driver Stretch Component Builder for assembling driver stretches, to Driver Part Component Builder for making driver parts, and finally back to Driver Duty Component Director for assembling the final product: a complete driver duty.

- **Strategy**
  Driver Duty Director is a Strategy pattern [Gamma+95, p. 315]. Driver Duty Director objects are the clients of the Component Builder objects. Each Director object is ultimately

responsible for building all the components in its layer of the cascade. This is achieved by directing all the builders within that layer to build their respective components, which are then assembled. Some of the responsibility for sequencing the leaf builders in any one layer is given to the composite builder at that level. For less complex construction processes, more of this responsibility for sequencing can be retained at Director level (i.e., omit the child relationship between a composite builder and its parent). The Driver Duty Constructor class is the context for this Strategy pattern.

- **Vehicle Block**
  Vehicle Block (to be written) represents blocks of vehicle work in a vehicle schedule. Vehicle blocks are partitioned into units of work that are assigned to driver spells.

- **Driver Duty Scheduler**
  Driver Duty Scheduler is the overall system within which Driver Duty Constructor resides and is the client of Driver Duty Constructor (Figure 9). Driver Duty Constructor is triggered into action by calls from Driver Duty Scheduler, a Facade object that provides a front for the entire Driver Duty Scheduler system. The detailed interactions between Driver Duty Scheduler and all the other subsystem classes to which it has access will be discussed in another paper (to be written).

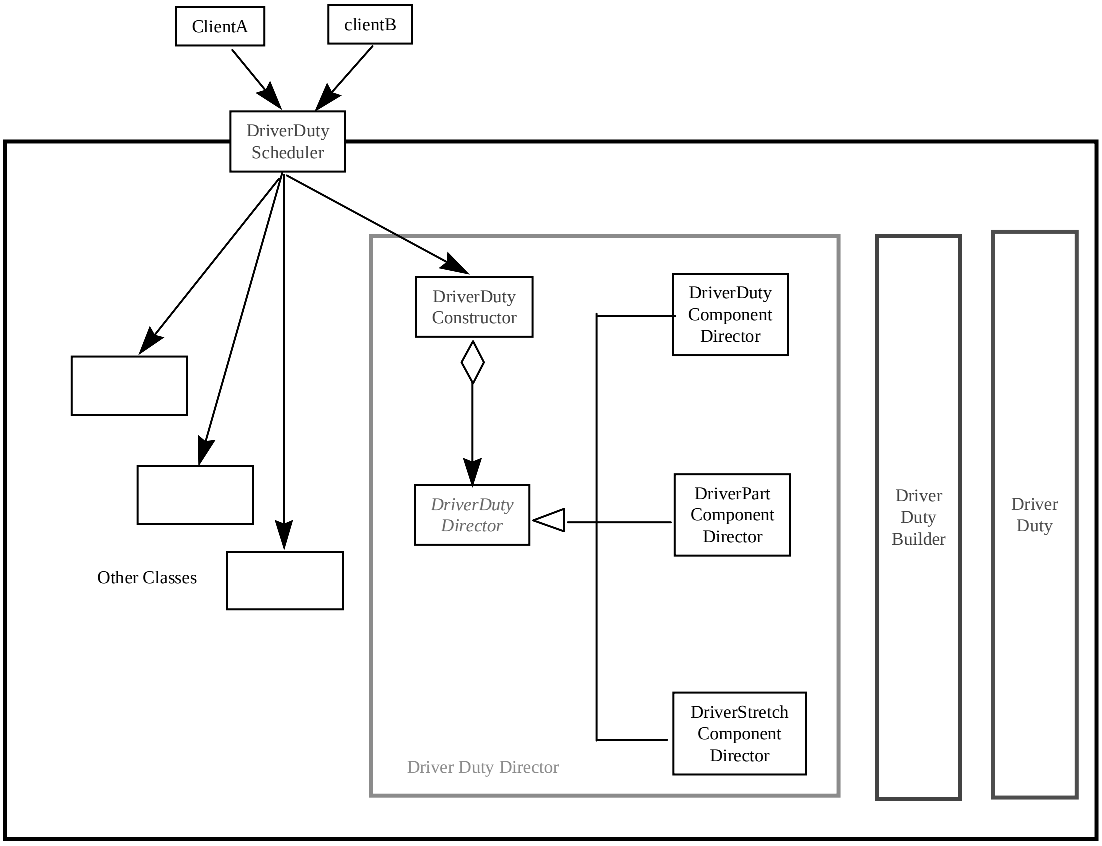

Figure 9. Driver Duty Scheduler.

Forces and Design Rationale

"Parts explosions are the most compelling examples of aggregation" [Rumbaugh+91, p. 59]. The relationship between a driver duty and its components suggests an aggregation tree, as shown in Figure 10. With this part-whole structure, we can represent a variety of different straight and split duties.

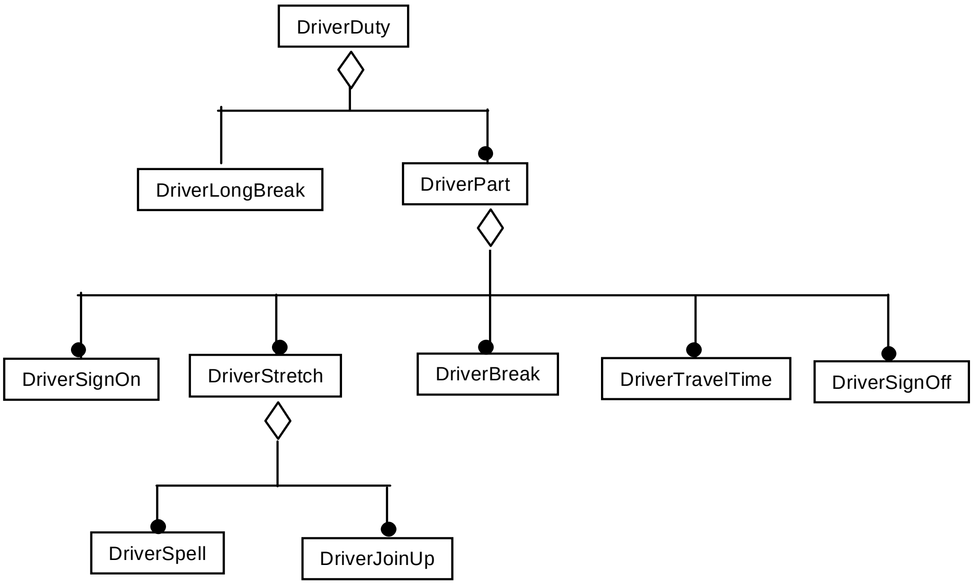

Figure 10. The part-whole relationship between a driver duty and its components.

However, this part-whole representation has a problem, i.e., clients have to treat duty components and their compositions differently. Gamma et al [Gamma+95, p. 163] pointed out that having to distinguish primitive objects and their compositions makes the application more complex. They proposed the Composite pattern [Gamma+95, p. 163] to overcome this problem. With Composite, we can define consistent interfaces for both duty components and their compositions and treat these objects consistently, as shown in Figure 11.

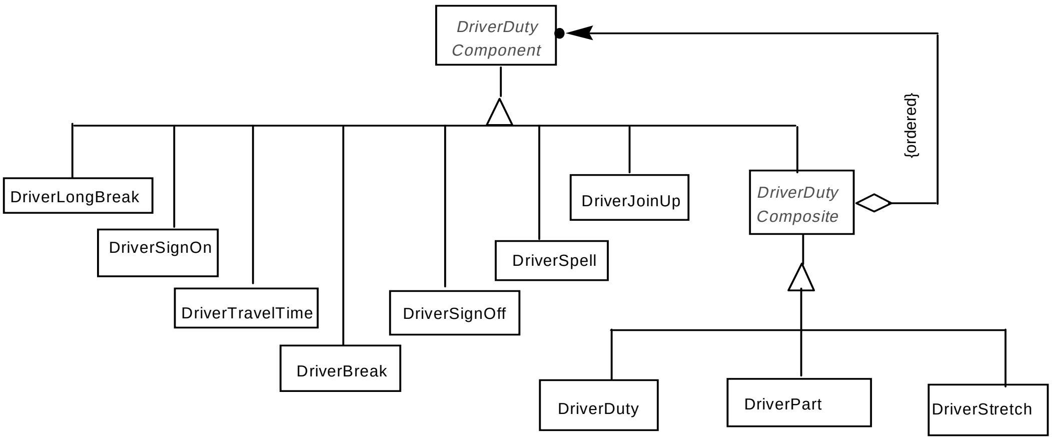

Figure 11. Driver Duty represented as a Composite.

So why do we need Cascade?

In Composite, different types of composite are supported via decomposing or sub-classing the Composite class; all the primitive objects are treated at the same level. For example, in Figure 11, Driver Duty Composite is decomposed into Driver Duty, Driver Part, and Driver Stretch. This decomposition introduces a problem; it cannot represent layering and ordering of objects and their compositions explicitly. We can represent this layering and ordering information in code, but in doing so we lose some of the value of using patterns as a language for communication with others. We want to show this layering and ordering information explicitly, so we developed the Cascade pattern.

In Cascade, we can layer different types of composite so they appear at different levels in different orders (Figure 4). We can express explicitly that a driver duty is made up of driver parts; a driver part is made up of driver stretches; a driver stretch is made up of driver spells. Such layering and ordering information is very important in the transport domain.

Cascade may require additional coding and this is the trade off for a more explicit specification. However, code is only a small part of the software engineering process. Achieving a clearer specification for discussions with domain experts and programmers is also a worthwhile goal for patterns.

Our Composite within Cascade as applied within Driver Duty differs from the Gang-Of-Four's Composite in that for any one layer, our Composite defines an ordered sequence of leaves and not a tree; one of the leaves is an abstract Component class for the next layer down (e.g., Driver Part Component and Driver Stretch Component). Our Composite never has another Composite as a child, except indirectly through an abstract interface with another Composite. In this way we treat any one Composite layer as a primitive component of its parent layer.

Why should Driver Duty Builder be another Cascade?

In our TOPS, we consider it important to focus not only on the process by which our objects achieve their purpose within themselves, but also on the process by which they interact with other objects to achieve their combined higher purpose. Organising the process for constructing a driver duty symmetrically with its structure (Figure 5) can help us to achieve this aim. We can delegate the responsibilities for controlling and co-ordinating parts of the driver scheduling process to appropriate objects and still maintain a reasonable view of the overall process. Under such symmetric structures, each builder is responsible for building each duty component and builders at higher levels can reuse their lower-level builders. In that way we can build many variations in duty as represented by the Driver Duty pattern. "A delegated style ideally has clusters of well defined responsibilities distributed among a number of objects. A hierarchy of control may be evident … A delegated control architecture feels like object design at its best." [Wirfs-Brock95].

Flexibility and reusability are major concerns in our design as our goal is to build software that can evolve and adapt like a living system. "Living systems are organised in such a way that they form multileveled structures, each level consisting of subsystems which are wholes in regard to their parts, and parts with respect to the larger wholes… All these entities — from molecules to human beings, and so on to social systems — can be regarded as wholes in the sense of being integrated structures, and also as parts of larger wholes at higher levels of complexity. In fact, we shall see that parts and wholes in an absolute sense do not exist at all." [Capra82, p. 27]. In Cascade, we can express the relationship between parts and wholes naturally. Builders create parts (i.e., the leaf objects), but the whole product for a particular layer is simply a part for the next layer up (Figure 5). *Wholes and parts are relative.* Driver Stretch is a whole whose main parts are Driver Spells; Driver Part is a whole whose main parts are Driver Stretch Components; Driver Duty is a whole whose main parts are Driver Part Components. The Cascade pattern allows its parts to preserve their individual autonomy and at the same time facilitates their togetherness.

We find patterns and symmetry in patterns beautiful. "The concept of symmetry is a familiar and important one in daily life. Many human products are deliberately built to be symmetric, for either aesthetic or practical reasons." [Davies95, p. 58]. Beauty implies good design. "The aesthetics of the design itself are good indicators of system maintainability. A system that can't easily be understood can't easily be evolved. Good design appeals to the human aspects of development" [Coplien96, p. 39].

Why should Driver Duty, Builder and Director be presented together?

Driver duty builders and directors are inseparable because they combine to construct a whole product; they are one. However, they have slightly different responsibilities in that builders build parts and directors assemble wholes from parts; they are two. *Wholes and parts are inseparable*. Directors are responsible for higher level products made up of lower level products constructed by builders; directors and builders are the top and bottom of a delegation of responsibility continuum.

Wholes and parts are relative, and so are the objects responsible for constructing them. Their roles overlap according to the context. For example, in our Driver Duty Builder and Director patterns, we have delegated the responsibility for co-ordinating a whole sequence of building tasks to builders. An object is not a director or a builder; it can be a bit of both. In some cases where clients want to interact with the system frequently and have a finer control over the construction process, we can simplify the builders and leave more of the ordering tasks to directors.

Similar Patterns and Applications

The Driver Duty and Driver Duty Builder Patterns are domain applications of the Cascade pattern [Foster+97]. Cascade has also been used to structure Route [Zhao+96], Point and Links [Foster+97] and Journey Variant [Foster+97]. Driver Duty Builder and Director work together as in the Gang-of-Four's Builder pattern [Gamma+95, p. 97]. Driver Duty Director is expressed as a Strategy pattern [Gamma+95, p. 315].

FINALE

Driver Duty Constructor is configured by the client with a concrete Director object; it is the context object for this Strategy pattern. It is triggered into action by calls from Driver Duty Scheduler, a Facade object that provides a front for the entire system. Building driver duties is a complex process. Some clients do not care about the details of duty construction; they simply pass on a request and leave the system to generate duties automatically. Others, on the other hand, want to control each step and build duties interactively. Driver Duty Scheduler and other classes are responsible for these diverse requirements. The detailed interactions between Driver Duty Scheduler and all the other system classes to which it has access will be discussed in other papers.

ACKNOWLEDGEMENTS

We wish to thank all the people involved in various Science and Engineering Research Council (UK) and European Union (EU) funded projects and others with whom we have discussed the problems of modelling public transport operations; also our two students at RMIT University (Australia), Lance Dourlay and Bradley Kazazes, for implementing Driver Duty and Driver Duty Builder during their vacation. Special thanks are due to Linda Rising, our PLoP shepherd, for her encouraging and insightful comments on the paper.

REFERENCES

[Capra82] F. Capra. *The Turning Point.* Simon & Schuster, New York, 1982.

[Coplien96] J. O. Coplien. *Software Patterns: A White Paper.* SIGS Publications, New York, 1996.

[Davies95] P. Davies. *Superforce.* Revised Edition Penguin Books, England, 1995.

[Foster+96] T. Foster and L. Zhao. "Modelling Transport Objects with Patterns". *Presented at EuroPLoP96,* 1996. To appear in *Journal of Object-Oriented Programming* and *Object Expert* respectively, 1997.

[Foster+97] T. Foster and L. Zhao. "Cascade". To submit to *PLoP97,* 1997.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[Grigg97] R. Grigg. *The Tao of Being.* Element, 1997.

[Rumbaugh+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Wirfs-Brock95] R. Wirfs-Brock. *Responsibility-Driven Design : Characterising Your Application's Control Style.* SIGS Publications. Report on Object Analysis & Design. Volume 1. No. 3 September-October 1994.

[Zhao+96] L. Zhao and T. Foster. "A Pattern Language of Transport Systems (Point and Route)". *Proceedings of PLoP96,* 1996. To appear in *Pattern Language of Programming Design 3.* Addison-Wesley, 1997.

[Zhao+98] L. Zhao and T. Foster. "A Pattern Language of Transport Systems (Driver Duty)". To appear in *Journal of Object-Oriented Programming,* 1998.