Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 19 (2013) 380 – 388

Procedia Computer Science

The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013)

Extensible Software Design of a Multi-Agent Transport Simulation

D. Grether, K. Nagel

Transport Systems Planning and Transport Telematics, Berlin Institute of Technology (TU Berlin), Salzufer 17-19, D-10587 Berlin, Germany, Tel: +49-30-314 21383, Fax: +49-30-314 26269, E-mail: grether@vsp.tu-berlin.de,

Abstract

This paper explains the chosen methodology for software design of the Multi-Agent Transport Simulation, MATSim. The design focusses on standard architectures and design patterns to ease usability and improve extensibility of the software. Potential for extension is discussed using an example implementation of a traffic signal control module. Both, MATSim and the extension are using the same concepts for software architecture.

Keywords: multi-agent simulation, transport simulation, software design

1. Introduction

Generally speaking, methodology for object-oriented software design is available and well documented at a lot of places, e.g. see the books of [1, 2, 3]. In the past decades, several concepts and considerations were established in software design — *Modularity*, *Information Hiding*, *Abstraction*, *Reusability*, *Usability* or *Extensibility* are prominent examples. These goals are partially competitive. In respect to software design, this implies some trade-offs and design decisions that are unique for each piece of software. In the following, trade-offs and methodologies of software design for the software named MATSim, i.e. *M*ulti-*A*gent *T*ransport *Sim*ulation, are discussed.

Multi Agent Transport Simulation and its acronym MATSim stand for strong words and concepts associated to them. For each of them, agent-based (transport) models, transport simulation and forecasting models many books and publications exist. In this paper, MATSim is seen as a software that enables usage of default functionality to answer transport analysis and forecasting problems on a high, agent-based resolution but for large scale applications. Large scale implies some level of abstraction to enable computationally cheap simulation and problem solving. Multi-agent based transport simulation are often implemented separating a simulation of physics from a simulation of agents. Agents interact with the physical world, e.g. as travellers in the transport system, or as control algorithms for traffic light control. Interaction may change certain aspects of the real world modelled. Yet, certain physical constraints should not be violated; for example, synthetic persons should be required to consume time while physically move from one location to another rather than being teleported in zero time (which is possible in the computer but not in reality).

doi:10.1016/j.procs.2013.06.052

Developed by distinct groups of developers, MATSim is the work of many; main developers are located at ETH Zurich, senozon AG, and TU Berlin. Even more heterogeneous is the group of users. In consequence there are several, distinct, domain-specific perspectives on the software.

Users of MATSim typically come from transport and civil engineering domains and are heading to solve problems for which the agent-based modelling approach is more suited than more traditional four step process modelling. The typical use-case starts with the collection and preparation of data. Then several simulations are run on a cluster for several days and compared afterwards. Often, this must be repeated several times as data may contain errors that do not become visible until simulation results are retrieved. If no automatic calibration is available, repetition of simulation runs might be needed to calibrate the model. Multi-agent simulation pays off when heterogeneous user preferences, time dependent user reactions or/and microscopic modelling of public transport are studied, e.g., see [4, 5, 6]. At the time these studies were undertaken, there was no default software support for the required functionality; MATSim had to be extended by custom model components. Today, some of this functionality is available in MATSim or as a contributing project. However, modelling problems on this high level of detail makes it hard to define a default methodology and implementation that suits every user. So, quickly a user finds himself in the second group, the researchers.

Advanced users may reach a point at which the model implemented in MATSim is not sufficient to answer their questions. Then, they start researching extended or distinct modelling approaches. These models shall be applied and implemented to MATSim, frequently. Addition of functionality to the basic algorithm or changes to small parts of it are sufficient in most cases. Yet, researchers must be able to write their model code. A strong computer science background is helpful, but can not be expected. Currently, such work is done e.g. in the field of evacuation [7] or destination choice of travellers [8].

From a computer science perspective MATSim can be seen as substrate delivering plausible, network wide traffic patterns that can be used to test agent-based concepts, e.g., see [9]. Furthermore computer science may provide programming languages and methodology that makes extension of the software as simple as possible. Choosing the most suitable approach, set-up of methodology and core concept implementation is the task of computer engineering. Optimally, this should be invisible to other users.

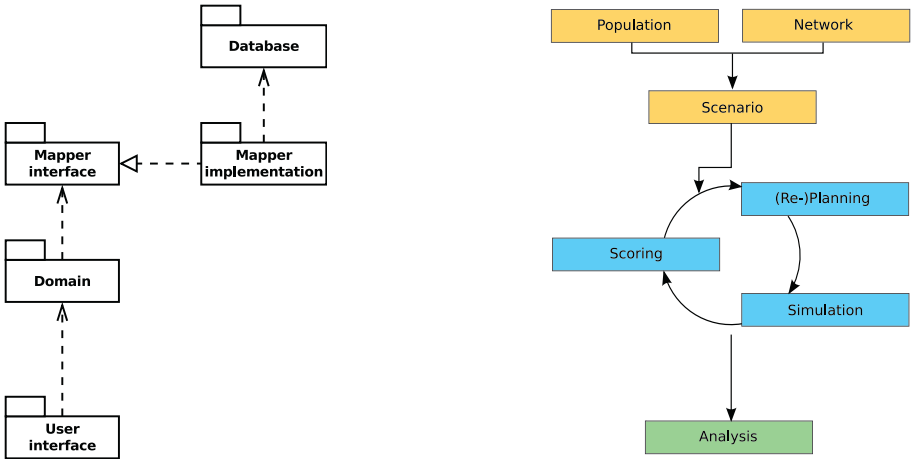
Unfortunately, invisibility is not reached. This paper explains which methodology and set-up was chosen in the last years in order to improve the first prototypical version of MATSim from 2007. Motivated by the domain-specific perspectives on the software the main goals are *Modularity*, *Reusability* and *Extensibility*. As most users come with a non computer science background practical solutions for achievement of goals are needed. Full knowledge of [1, 2, 3] can not be expected, thus a subset of these concepts is required. This paper reviews the chosen subset. Two software design patterns, i.e. *Abstract Factory* and *Observer* [1], in conjunction with some other standard software design approaches improve the main design goals. As example for an extension of the MATSim core algorithm the paper shows how a traffic signal implementation can be attached to MATSim.

The rest of the paper is organized as follows: In Sec. 2 the current design of MATSim is explained, while Sec. 3 shows how an extension for the simulation of traffic signals is designed. The paper ends with a discussion and conclusion.

2. Software Design of MATSim

In this section, first the central algorithm of MATSim is reviewed, followed by a description of overall sofware design. For the different parts of MATSim potential extensions are discussed.

Theory and Algorithm. In MATSim, each traveler of the real system is modeled as an individual virtual person. The approach consists of an iterative loop that has the following important steps: 1. *Plans generation*: All virtual persons independently generate daily *plans* that encode, among other things, their desired activities during a typical day as well as the transportation mode. Virtual persons typically have more than one plan ("plan database"). 2. *Traffic flow simulation*: All selected plans are simultaneously executed in a simulation of the physical system (often called "network loading"). 3. *Scoring*: All executed plans are scored by a *utility function* which can be personalized for every individual. 4. *Learning*: At the beginning of



(a) Typical layered Architecture, source [12]

(b) A layered architecture for MATSim

Fig. 1: Overall software architecture

every iteration, some virtual persons obtain new plans by modifying copies of existing plans. This is done by several *modules* that correspond to the choice dimensions available, e.g. time choice, route choice, and mode choice. All other virtual persons choose between their plans according to a Random Utility Model (RUM).

The repetition of the iteration cycle coupled with the plan database enables the virtual persons to improve (learn) their plans over many iterations. This is why it is also called *learning mechanism* which is described in more detail in [10]. The iteration cycle continues until the system has reached a relaxed state. At this point, there is no quantitative measure of when the system is "relaxed"; users allow the cycle to continue until the outcome is stable.

Overall Architecture. The overall architecture of the first release of MATSim in 2007 was rather fuzzy. Coupling was high and modularization at a preliminary stage. Complex code inheritance hierarchies and global variables persisting the state of modules reduced *Reusability* and *Extensibility*. Especially *Hybrids* [3, pp. 93], i.e. objects that possess properties of data structures and provide functionality, reduced *Modularity*. A more detailed discussion of issues can be found in [11].

Layered architectures, "that should be familiar to anyone who works with information systems" [12] can help to reduce coupling, improve modularization and sort out dependencies between modules. The typical overall layer structure is shown in Fig. 1a. It consists of a Database, a Mapper implementation accessing the concrete Database, an abstract interface for Mappers, the Domain to be modelled in the software and a User interface. This typical layered approach is applied to MATSim. Fig. 1b shows a colored version of the control flow of MATSim that is used for documentation at many places. In yellow, the mapper to the Database is depicted, blue elements represent the main modules of the Domain while green is the User interface layer. Note that the arrows don't show dependencies but are oriented at the general course of action and thus may point into opposite direction than arrows in Fig. 1a. Dependencies between the layers are as shown in Fig. 1a.

Motivated by the considerations in [11], the software design for MATSim uses mainly two software design patterns from [1]. First, *Abstract Factories* are provided that construct the central components of the software and force use of Interfaces instead of implementations. Second, in order to make control flow and object communication translucent, *Observer* is frequently used.

Database & Mapper Layer. Microscopic transport simulation requires lots of data. None of the institutions participating in the development focusses on data modelling, nor is expert knowledge for modelling data ubiquitous at the developer side. Due to this consideration, the amount of data structures required by the

core algorithm are reduced to a minimum: An abstract graph representation of a transport network, and a population of virtual persons that are travelling within the transport network. For both data structures, corresponding mapper interfaces and implementations are provided. These can be added to a mapper container, the `Scenario`. Additionally, access to the mapper for the configuration is provided in `Scenario`. Within the iteration cycle, access to the data is granted via `Scenario`.

For many studies, more data is needed than provided by network and population. Data sources are heterogeneous and may vary from study to study. In contrast, a manually programmed simulation logic must be encoded against specific concepts derived from data. Effort of implementation is reduced if these concepts can be accessed in a type-safe manner. Maintenance cost increase significantly and extensibility is reduced if central parts of the central algorithm need to parse *String*s that have to be well formed. Thus, modules that extend the core simulation functionality are expected to implement their own data mappers. These can be added to `Scenario` as shown in Listing 1.

```
public void addScenarioElement(Object o);
public boolean removeScenarioElement(Object o);
public <T> T getScenarioElement(Class<? extends T> klass);
```

Listing 1: Extension of `Scenario`: Signature

The Signature of the `getScenarioElement(..)` method may look a bit scary, but usage is simple, as shown in Listing 2: A custom data mapper of type `MyData` is added to an instance of `Scenario`. Later it is retrieved via the `getScenarioElement(..)` method in a type-safe manner.

```
MyData data = new MyData();
scenario.addScenarioElement(data);
...
MyData data2 = scenario.getScenarioElement(MyData.class);
```

Listing 2: Extension of `Scenario`: Usage

Domain: Simulation, Scoring and Replanning. The central algorithm of MATSim consists of three steps: Simulation, Scoring and Replanning. As indicated by Fig. 1b, each step is associated with a separate module. In 2007, the functionality of these modules was rather limited, while the number of users willing to extend the software was growing fast. Thus, a simple method for functional extension of these modules was required. The control flow and communication between these modules, however, should become invisible to enforce modularization.

In order to hide implementation details of the core modules, Java interfaces are extracted from the concrete implementations of the 2007 prototype. Use of these interfaces is encouraged by *Abstract Factories* creating the modules. On this top-level view on the domain, interfaces can be rather abstract. E.g. Listing 3 shows the top-level interface that a mobility simulation needs to implement in order to work within MATSim, while Listing 4 shows the appropriate interface of the factory creating the mobility simulation.

```
public interface Mobsim {
  public void run();
}
```

Listing 3: Top-level interface for a mobility simulation

```
public interface MobsimFactory {
  public Mobsim createMobsim(Scenario sc, EventsManager eventsManager);
}
```

Listing 4: Top-level factory to create a mobility simulation

This enables users to replace the mobility simulation completely if their requirements are not met by the default implementation.

In order to hide control flow and communication between modules, the *Observer* pattern is used at two places:

- First, the mobility simulation uses the same *Observer* implementation as found in the first 2007 release. Each time, something is happening in the simulation, e.g. a virtual person departs for a trip, or a vehicle enters a new link of the transport network, an `Event` is generated and sent via the class `EventsManager` to all `EventHandler` instances registered for the concrete event class. Essentially this is an *Observer* implementation for the mobility simulation where the simulation is the *Model* and *Controller* while `EventHandler` is the top level interface for the *Views*. The mobility simulation retrieves access to an `EventsManager` when it is created via the factory method, see Listing 4.
- Second, new functionality to the steps of the central algorithm can also be added via an *Observer*: The state of the iterative relaxation process represents the *Model* element of the pattern. States are e.g. startup and initialization completed, the start of an iteration, the end of microsimulation execution, the end of the scoring or the end of the iteration cycle[1]. The *Controller* element of the pattern, that alternates the states, is the central algorithm of MATSim (currently implemented in the `Controler` class). Users can provide custom implementations for the *View* elements of the *Observer* pattern. This allows addition of functionality at each point of the iterative process. E.g. code for analysis could be added at the end of each iteration without touching any code concerning setup and control flow of the whole simulation process.

The combination of *Abstract Factories* and *Observer* makes the central algorithm extensible, as users can connect to the core algorithm at each step adding their specific *View* implementation providing any additional functionality required. If the core modules do not fit needs, they can be replaced completely. Thus, the overall process is considered to be flexible and extensible. However, replacing a whole module of the core algorithm, as e.g. a complete replacement of the mobility simulation, can result in much effort. Therefore, well designed default models and implementations are needed, which are also suited for extension and customization.

Default Models & Implementations. Obviously, specifications of abstract interfaces can not be run as there is no implementation. Thus, there is a certain amount of default implementations that can be used and configured via the configuration file of the simulation. If a user states he uses MATSim, he should refer to the default implementation. His specific set of configuration options should be documented, if non-default parameters are used. If users add or modify code, it is expected that code modifications are also documented and published. In order to reduce documentation effort and amount of code to be rewritten, certain parts of the default implementations may be reused. To ease effort, in an ongoing debate and refactoring process, default implementations are hidden behind abstract type specifications and interfaces suitable for customization.

If Delegation and the Patterns *Abstract Factory* and *Observer* are used consequently, extensibility can be realized up to a certain extent [11]. In addition, the resulting software architecture can be understood by knowledge of the two patterns and the concept of delegation. E.g. the default production mobility simulation of TU Berlin (referred as `QSim`) uses the *Observer* pattern twice: First, the `Event` mechanism can be used by *Views* to get informed about things that happen in the simulation. A second *Observer* is provided for the control flow of the mobility simulation. E.g. one can register a *View* that is informed when the mobility simulation is set up, incremented or shut down. By use of a *Factory* creation of objects for virtual persons or other agents fed into the mobility simulation can be changed. Default behaviour of virtual persons thus can be customized. This repetition of concepts and patterns results in a Matryoshka Doll. Each time one of the modules of overall simulation is unfold, same concepts appear, thus reducing overhead to understand design concepts before extensions and modifications can be provided.

Provided that existing source code should not be touched functional extension is possible by use of patterns as *Abstract Factory* and *Observer* in conjunction with the OO language concepts of Java. However,

[1] see `www.matsim.org/node/602` for the official documentation, last access 17.01.2013

extensibility is also restricted by this subset of programming concepts. Restrictions can be decreased by use of patterns as e.g. *Visitor*. The understandability of code, however, also decreases by more sophisticated patterns. Especially the addition of new operations to existing types is considered hard if the code should not be modified. A detailed discussion can be found in [11]. New operations can also be added as completely separated modules. This may reduce cohesion, but on the other hand leads to very modular extensions. The default modules require some interface definitions to communicate with the extensions. The *Observer* implementations provide this interfaces. The next section shows the design of such an extension for a module to simulate traffic signal control.

3. Traffic Signals Extension

Traffic signal control is an important element of urban transport systems. For the purpose of transport forecasting and simulation, the inclusion of traffic signal control may be desired, but because of the need of extensive amounts of data, it may not be feasible.

Fixed-time and traffic-actuated signal control are probably the most frequently used traffic signal control strategies. In contrast, using "agents" for traffic signal control is subject of current research. The agent paradigm here typically refers to agents that control all traffic signals of a signalized junction. The specific type of agent, e.g. rule-based or learning, in the sense of [13] is not specified. Such a wide use of the agent paradigm covers all approaches for traffic-signal control as even a fixed-time control can be seen as a rule based agent. An agent for traffic signal control needs to be able to capture the state of its environment, e.g. the number of cars approaching the junction controlled by the agent. Furthermore, the agent is responsible to compute a state of traffic signalization that should be conform to legal requirements. Finally, it should result in a good performance of traffic flow.

In the following, the design and implementation of a software architecture is proposed that allows agent-based modelling of traffic signal control as an optional component of MATSim. Software design and implementation follow the design patterns used in the core of MATSim. Instead of being modelled as extensions to the MATSim type hierarchy, the model is a stand-alone extension that is coupled to MATSim via one concrete interface declaration that must also be implemented by the core of the simulation framework. It was possible to solve all other coupling by using the *Abstract Factory* or *Observer* implementations reviewed in the previous section.

The traffic signal module for MATSim consists of three components: A data mapper providing access to a database, a component that plugs the module into MATSim, and a default implementation for fixed-time control. The latter also makes the implementation of further models for traffic signal control easier.

Data. For microscopic simulation of traffic signals, minimally a description of the real world traffic signals is needed, i.e. at which locations in the network traffic signals are located and which edges or turning moves of the network are influenced by them. Individual traffic signals are often assigned to groups; each traffic signal of a group shows the same color all time. In addition, the data format should allow to specify the control strategy used for sets of groups. Optionally, one should be able to specify legal constraints. This includes the time of amber or red-amber a traffic signal needs to show before it can switch to red or green. Intergreens specify the time signals have to be all red before a signal can switch to green. All this data is provided by a mapper interface to a database called `SignalsData` providing access to the subcomponents. The mapper is added to the `Scenario` and is available at relevant places within the simulation process. The `SignalsData` mapper and its child containers are behind interfaces to ensure that the current XML Schema based data format can be replaced by something else.

Default Model & Implementation. Several components are identified that can be used to model and microsimulate traffic signals. Fig. 2 shows types of the MATSim core modules from the package `org.matsim` (blue). Furthermore, main types of the traffic signal extension are shown with their dependencies to the core modules. These components and dependencies are reviewed in the following.

First, a component that represents the real world infrastructure is required. It serves as interface to a mobility simulation, i.e. it communicates the color of traffic signals to links of the transport network. The

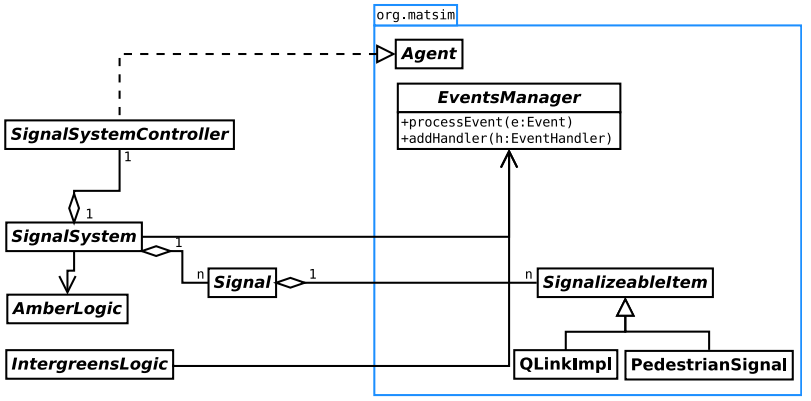


Fig. 2: Components of the default signal systems module and their connection to MATSim

communication is realized by the interfaces `Signal` and `SignalizeableItem` of Fig. 2. The latter is the interface within `org.matsim`. It has to be implemented by mobility simulations in order to support traffic signals. Two currently available example implementations are `QLinkImpl` and `PedestrianSignal`.

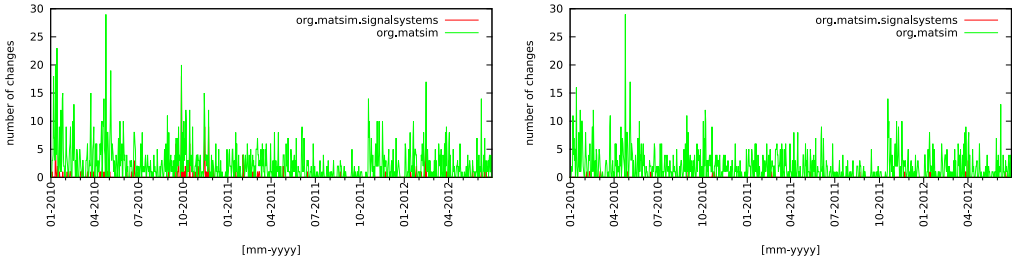
The second required component provides traffic signal control strategies for signalized intersections. For real world applications, it should be possible to control each signalized intersection with a different control strategy. For this, the `SignalSystemController` interface of Fig. 2 can be used. This component can be seen as agent if an agent-paradigm is used. Via the `SignalSystem`, the control strategy can access and switch the state of the infrastructure representation, i.e. the `Signal` instances. Furthermore, access to the `EventsManager` instance of `org.matsim` is provided in order to feed changes of signalization back to the central `Events`' *Observer* implementation.

The third component provides a tool for control strategies to reduce the amount of implementation in control strategies for switches between red and green light. In between, colors red-amber and amber may be shown for a certain time. The time depends completely on legal constraints which, in turn, may depend on the layout of the crossing or on the speed-limit. The `AmberLogic` interface is accessible via `SignalSystem` and receives requests to change color between red and green and translates them into a color sequence that shows also the required time of red-amber and amber light.

The last component is a logic that checks if conflicting approaches of a junction got a right-of-way at the same time, producing a warning or throwing an error in such a case. This component is necessary since none of the current mobility simulations available within `org.matsim` represents vehicle collisions. Thus, this component is quite important as validity of the control strategy can be ensured. The `IntergreensLogic` in Fig. 2 provides this functionality. It is a *View* on the `EventsManager` of `org.matsim` and thus completely decoupled from other components of the traffic signal extension.

Integration into MATSim. The *Observer* implementation of the top-level simulation process is used to plug the traffic signal extension into MATSim. After notification that the simulation is at start-up, the signals data is loaded, and the simulation model for traffic signal control is set up. At the beginning of each iteration, the model is triggered to reset its components, and at the end of simulation some data is persisted. The *Observer* for the control flow of the mobility simulation is used to update the state of the traffic signal control during execution of mobility simulation.

As traffic signals are considered important but still optional for transport simulations, a central factory at the top level of the domain layer is added. Using this factory, the complete module can be exchanged easily, or the default implementation can be retrieved for further customization of its components. Each component can be exchanged or customized via factories.



(a) Changes per day

(b) Changes per day, without some bugfixes

Fig. 3: Changes on package org.matsim and org.matsim.signalsystem

4. Discussion

Overall Design of MATSim. The overall software design of MATSim was improved since the first release in 2007 appeared. All global singletons were removed from MATSim core components. Hybrids are vanishing because of the layered architecture. Extensions to standard input data can be added to the `Scenario`. The mechanism described in this paper can also be found in the API offered by the `ApplicationContext` of the *spring*[2] framework. Thus, this appears to be standard functionality for extensible Java software. As *Design Patterns* mainly *Abstract Factory* and *Observer* are used. Usage of these two patterns can be understood relatively easy, especially when comparing to other patterns e.g. *Visitor*. Furthermore, they are the standard solution for many problems, and are probably two of the best known and most frequently used patterns in OO languages.

Modularity of MATSim was improved because of the use of Java interfaces in conjunction with *Abstract Factory* and *Observer*. The patterns encourage use of interfaces instead of implementations. The existence of a Java interface communicates a certain level of design thoughts for a module. The mental barrier of developers to jump over the hurdle to change an interface seems to be higher than just to change implementation. Thus, the development towards principle software design goals and especially *Modularity* is encouraged.

Using standard Java, *Delegation*, *Abstract Factory* and *Observer Extensibility* can be provided up to a certain extent without changing core components. But the approach is limited. *Abstract Factory* implementations using delegation get hard to understand if several modules are interlaced and used together. *Observer* lacks a definition of the sequence in that *Views* are informed about changes of the *Model*. This might be seen as drawback, but is actually the quintessence of *Observer* application – encapsulating functionality in modules that can be used independently and thereby reducing control flow between them to a minimum. The *Model* object informs the *View* objects in a not defined sequence about changes, i.e. *"you don't want these objects tightly coupled"* [1, p. 294]. This strong decoupling results in plugins that don't depend on each other.

Traffic Signals Extension. MATSim now comes with a traffic signal module containing a default implementation for fixed-time traffic signal control. The paper shows how the module can be attached to MATSim with the current software design. The module requires the mobility simulation of MATSim to implement a single interface (`SignalizeableItem`). The architecture of the module follows the same concepts as the core software in order to reduce complexity. All communication between the MATSim core and the traffic signal module can be realized by use of the different *Observer* implementations.

All components of the traffic signal module can be customized, replaced or extended without changing the default implementation or core components of MATSim. Due to the use of *Interfaces*, *Delegation* and

[2] `www.springsource.org/`, last access 19.03.2013

Abstract Factory each component can be exchanged separately. The default traffic signal module provides an implementation for fixed-time traffic signal control. In [14] an extension modelling a traffic-actuated signal control strategy was added by only implementing the algorithm for traffic-actuated control and the code building the software. For the representation of real world infrastructure, computation of amber times, and coupling to MATSim core modules code from the default implementation is used. Thus, the chosen architecture seems to have the required flexibility for extensions.

Maintenance of the implementation was quite cheap during the last years. Fig. 3a shows in green the changes recorded by subversion to all code within the package `org.matsim` between 2010 and 2012. Changes to the package `org.matsim.signalsystems`, which contains the model specification and the fixed-time control implementation, are shown in red. While `org.matsim` in total was changed quite frequently, changes to `org.matsim.signalsystems` are rare. This means that for all classes within `org.matsim.signalsystems`, coupling is low. The changes to the signal systems package occurred mainly due to feature improvements and bug fixes by the developer responsible for the extension. E.g. during the period 2010–2012 a feature for visualization was added to the package, also the intergreen logic was developed. Fig. 3b shows same information as Fig. 3a, but removes all changes due to this bug fixes and improvements. Red changes nearly disappear, reinforcing the argument that coupling is low.

Overall, this shows that the chosen approach is modular, uses few patterns, but is extensible.

5. Conclusion

This paper explains the chosen methodology for software design of the Multi-Agent Transport Simulation, MATSim. The design focusses on standard architectures and design patterns to ease usability and improve extensibility of the software. Potential for extension is discussed using an example implementation of a traffic signal control module. Both, MATSim and the extension are using the same concepts for software architecture. Reduction of concepts leads to a clear and in terms of OO software straight-forward architecture that reduces overhead of maintenance by enforcing modularization while extensibility is possible.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns. Elements of reusable object-oriented software, Addison-Wesley, 1995.
[2] M. Fowler, Refactoring: Improving the design of existing code, Addison-Wesley, 2004.
[3] R. C. Martin, Clean code. A Handbook of Agile Software craftsmanship, Pearson Education, Inc., Right and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 002116, USA, 2009.
[4] M. Rieser, U. Beuck, M. Balmer, K. Nagel, Modelling and simulation of a morning reaction to an evening toll, in: Innovations in Travel Modeling (ITM) '08, Portland, Oregon, 2008, also VSP WP 08-01, see www.vsp.tu-berlin.de/publications.
[5] D. Grether, B. Kickhöfer, K. Nagel, Policy evaluation in multi-agent transport simulations considering income-dependent user preferences, in: Proceedings of The 12th Conference of the International Association for Travel Behaviour Research (IATBR), Jaipur, India, 2009, also VSP WP 09-13, see www.vsp.tu-berlin.de/publications.
[6] A. Neumann, K. Nagel, Avoiding bus bunching phenomena from spreading: A dynamic approach using a multi-agent simulation framework, VSP Working Paper 10-08, TU Berlin, Transport Systems Planning and Transport Telematics (2010).
[7] G. Flötteröd, G. Lämmel, Evacuation simulation with limited capacity sinks – an evolutionary approach to solve the shelter allocation and capacity assignment problem in a multi-agent evacuation simulation, in: Proceedings of the International Conference on Evolutionary Computation, SciTePress, 2010, pp. 249–254.
[8] A. Horni, K. Nagel, K. Axhausen, High-resolution destination choice in agent-based demand models, Annual Meeting Preprint 12-1989, Transportation Research Board, Washington D.C., also VSP WP 11-17, see www.vsp.tu-berlin.de/publications (2012).
[9] A. Bazzan, Opportunities for multiagent systems and multiagent reinforcement learning in traffic control, Autonomous Agents and Multi-Agent Systems 18 (2009) 342–375, 10.1007/s10458-008-9062-9.
[10] M. Balmer, B. Raney, K. Nagel, Adjustment of activity timing and duration in an agent-based traffic flow simulation, in: H. Timmermans (Ed.), Progress in activity-based analysis, Elsevier, Oxford, UK, 2005, pp. 91–114.
[11] D. Grether, Software design of a multi-agent transport simulation, VSP WP 13-01, see www.vsp.tu-berlin.de/publications (2013).
[12] M. Fowler, Reducing coupling, Software, IEEE 18 (4) (2001) 102–104. doi:10.1109/MS.2001.936226.
[13] S. Russel, P. Norvig, Artificial Intelligence – A Modern Approach, 3rd Edition, Pearson Education, Inc., 1 Lake Street, Upper Saddle River, New Jersey 07458, USA, 2010.
[14] D. Grether, J. Bischoff, K. Nagel, Traffic-actuated signal control: Simulation of the user benefits in a big event real-world scenario, in: 2nd International Conference on Models and Technologies for ITS, Leuven, Belgium, 2011.