# villager: A framework for designing and executing agent-based models in R

**Thomas Thelen**[*1], **Marcus Thomson**[†1], **Gerardo Aldana**[‡2], **and Toni Gonzalez**[§3]

**1** National Center for Ecological Analysis and Synthesis **2** College of Creative Studies, University of California, Santa Barbara **3** Department of Anthropology, University of California, Santa Barbara

**DOI:** 10.21105/joss.04562

**Software**

- Review
- Repository
- Archive

**Editor:** Marcos Vital

**Reviewers:**

- @chrisaberson
- @brunomontezano
- @thodson-usgs

**Submitted:** 03 May 2022
**Published:** 10 November 2022

## Summary

Villager is an agent-based modeling framework: it prescribes a convention and interface for modelers to create and run agent-based models (ABM). The framework is aimed at researchers in the social sciences who are focused on modeling human populations. The key features of villager are:

1. Scalability: `villager` makes extensive use of the R6 class system (Chang, 2020), enabling the power of reference semantics without the hurdles of manual memory management. This enabled an architecture design where user-supplied functions are run within the framework. The reference semantics also enable cheaper memory operations by allowing for the mutation of agents in-place rather than costly copy semantics.
2. Extensibility: `villager` exposes a number of classes that can be extended by domain scientists to provide flexibility in experiment design. The extended classes can be "plugged" into the villager framework and run seamlessly.

Together, these two features allow researchers to design ABMs with flexible requirements-both functionally and computationally.

## Statement of need

Agent based modeling has found use in an increasing number of applications ranging from market dynamics, animal behavior, and population studies (DeAngelis & Diaz, 2019). There are only a few agent-based modeling systems available for researchers using R, a popular language among social scientists. In some cases researchers must bootstrap their own ABM systems due to lack of available packaging that provides flexible modeling. The most popular R ABM frameworks include RNetLogo and SpaDES. Although RNetLogo is powerful, it acts as an interface to the NetLogo software. This requires Java and pipes NetLogo syntax to the Java process rather than using native R to describe system dynamics (Thiele & Grimm, 2010). SpaDES supports agent-based modeling however, its primary use is for Discrete Event Simulations (McIntire et al., 2022). Villager differentiates itself from these two by being R native and specifically designed for flexible ABM simulations.

[*] Co-first author
[†] Co-first author
[‡] Co-first author
[§] Co-first author

## Functionality and design

### Modular

One of the main design goals was to keep the framework components separated in a modular fashion for long term maintainability to allow framework additions in the future. An additional goal was to present the smaller components to modelers in a way that allows for them to extend each part of the framework to their needs.

### Extensible

Villager is made up from a few core classes, shown in Table 1 below. Base classes are provided to contain basic functionality and are designed to be extended by modelers.

| Class | Role | When to Subclass |
|---|---|---|
| agent | A single agent with typical properties for human agents such as name, age, and sex. | When agents need to have additional properties defined for more context, such as dietary preferences, weight, and food production restrictions. |
| resource | An abstract thing that an agent can possess. It has a name and quantity. | When resources need to have more complex attributes such as expiration dates or possession histories. |
| data_writer | Responsible for managing the serialization of simulation data. | To connect with additional data sources or file formats. |

Table 1: A summary of the classes that users can extend.

The `agent` class provides all of the main properties for individual agents. Because it's unlikely that the included properties will fit every researchers' needs, this class can be subclassed to include any number of properties ranging from simple constructs like personal wealth to more advanced ideas such as memory and emotional state.

The `resource` class is an abstract *thing* that a agent possesses. The base model only includes information about the name of the resource and the associated quantity. Modelers can extend this class with additional properties such as expiration date, date acquired, or previous owners.

By subclassing the `data_writer` class, users have the ability to control how and where their model data is stored. Storage locations and formats can range from remote databases and local files such as CSV, SQLite, or Microsoft Excel spreadsheets.

## Usage

A simulation consists of three parts: an initial condition that defines the initial state, models that are run at each timestep, and an interface to the simulation that define sthe experiment duration.

### Initial Conditions

Initial conditions are defined by creating a function, defining the state inside of it, and attaching it to a village. Because the function is executed *before* any time steps, it sets the state at t=0. The initial condition function requires the following parameters.

1. `current_state`: A mutable copy of the state representing the current time step.
2. `model_data`: User supplied data that persists through the simulation.
3. `agent_mgr`: An object that manages the agent with convenience functions for retrieval and creation.
4. `resource_mgr`: An object that manages the resources with convenience functions for retrieval and creation.

For example, an initial condition of a population with three agents

1. A mother
2. A father
3. A daughter

```
initial_condition <- function(current_state, model_data, agent_mgr, resource_mgr) {
  mother <- villager::agent$new(first_name=”Kirsten”, last_name=”Taylor”,
     age=9125, profession=”Fisher”)
  father <- villager::agent$new(first_name=”Joshua”, last_name=”Thompson”,
     age=7300, profession=”Laborer”)
  daughter <- villager::agent$new(first_name=”Mariylyyn”, last_name=”Thompson”,
     age=1022, profession=”None”)
  daughter$mother_id <- mother$identifier
  daughter$father_id <- father$identifier

  # Connect the mother and father
  agent_mgr$connect_agents(mother, father)
  # Add them to the manager
  agent_mgr$add_agent(mother)
  agent_mgr$add_agent(father)
  agent_mgr$add_agent(daughter)
}
```

## Models

Models are functions that contain code that's executed at each time step. Similar to initial conditions, models require the following parameters.

1. `current_state`: A mutable copy of the state representing the current time step.
2. `previous_sate`: An immutable copy of the previous state.
3. `model_data`: User supplied data that persists through the simulation.
4. `agent_mgr`: An object that manages the agent which has convenience functions for retrieval and creation.
5. `resource_mgr`: An object that manages the resources which has convenience functions for retrieval and creation.

Consider a model that prints the current step and increases the age of each agent by 1 at each time step and sets their profession to *Farmer* when they reach the age of 4383.

```
inc_age <- function(current_state, previous_state, model_data, agent_mgr, resource_mgr)
  print(paste(”Time step :”, current_state$step))
  for (agent in agent_mgr$get_living_agents()) {
    agent$age <- agent$age+1
    if (agent$age >= 4383 && agent$profession != ”Farmer”) {
      print(”Setting the profession to Farmer”)
      agent$profession <- ”Farmer”
    }
  }
}
```

## Simulation

Initial conditions and models make up the heart of the simulation. Villager aggregates these into *village* objects. The *simulation* object contains any number of villages inside. The example below uses the initial condition and model provided above to create a simulation that runs for 100 time steps. Because the agent behavior is scoped to a village, multiple villages may be defined-each having different agent dynamics.

```
small_population <- villager::village$new("Family Group", initial_condition, inc_age)
simulator <- villager::simulation$new(100, list(small_population))
simulator$run_model()
```

## Dependencies

Villager only depends on a few dependencies for core functionality.

| Package | Use |
|---|---|
| readr (Wickham & Hester, 2020) | Writing simulation states to disk. |
| uuid (Urbanek & Ts'o, 2020) | Generating unique agent identifiers. |
| R6 (Chang, 2020) | All villager classes are R6, allowing users to use reference semantics with models. |

## References

Chang, W. (2020). *R6: Encapsulated classes with reference semantics*. https://CRAN.R-project.org/package=R6

DeAngelis, D. L., & Diaz, S. G. (2019). Decision-making in agent-based modeling: A current review and future prospectus. *Frontiers in Ecology and Evolution*, *6*. https://doi.org/10.3389/fevo.2018.00237

McIntire, E., Chubaty, A., Luo, Y., Bauduin, S., Cumming, S., Marchal, J., & j7git. (2022). *PredictiveEcology/SpaDES: v2.0.8* (Version v2.0.8) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.6116101

Thiele, J. C., & Grimm, V. (2010). NetLogo meets R: Linking agent-based models with a toolbox for their analysis. *Environmental Modelling & Software*, *25*(8), 972–974. https://doi.org/10.1016/j.envsoft.2010.02.008

Urbanek, S., & Ts'o, T. (2020). *uuid: Tools for generating and handling of UUIDs*. https://CRAN.R-project.org/package=uuid

Wickham, H., & Hester, J. (2020). *readr: Read rectangular text data*. https://CRAN.R-project.org/package=readr